

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Introduction

Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- The primary databases used at Fidelity are:
 - Oracle
 - DB2
 - Microsoft SQL Server
 - Sybase
- All of these databases are relational databases
- SQL is the access language for all relational databases
- PL/SQL is unique to Oracle but similar to the procedural languages used by the other relational databases
- As an FSE, you will be writing and maintaining SQL and PL/SQL code

Course Outline

- Chapter 1 What Is Structured Query Language?
- Chapter 2 SQL Query Syntax
- Chapter 3 SQL Scalar Functions
- Chapter 4 SQL Joins
- Chapter 5 Additional SQL Functions
- Chapter 6 Data Manipulation Language
- Chapter 7 Databases with JDBC (Java Database Connectivity)
- Chapter 8 Updating Databases
- Chapter 9 Working with a Data Access Object
- Chapter 10 Advanced JDBC

Course Outline (continued)

- Chapter 11 Aggregating Information
- Chapter 12 Set Operators
- Chapter 13 Programming with PL/SQL
- Chapter 14 Creating Stored Procedures, Functions, and Packages
- Chapter 15 Testing PL/SQL
- Chapter 16 Creating Triggers
- Chapter 17 Data Definition Language
- Chapter 18 Scaling Relational Databases
- Chapter 19 Big Data and NoSQL
- Chapter 20 Amazon DynamoDB

Course Objectives

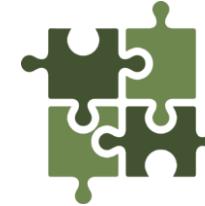
In this course, we will:

- Examine the role of SQL and the basic toolset
- Select, filter, and sort data from the database
- Manipulate the data with Oracle functions
- Extract data from multiple tables
- Aggregate data with the group functions
- Create and manage tables, views, and indexes
- Program with PL/SQL
- Create stored procedures, functions, and packages
- Create and work with triggers
- Explore data quality and data movement

Key Deliverables



Course Notes



Project Work



There is a Skills Assessment for this course

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 1: What Is Structured Query Language?

Chapter Overview

In this chapter, we will explore:

- The role of SQL
- Concepts of data modeling
- The course environment
- SQL Developer
- The Fidelity Development Environment

Chapter Concepts

What Is SQL?

Designing a Database

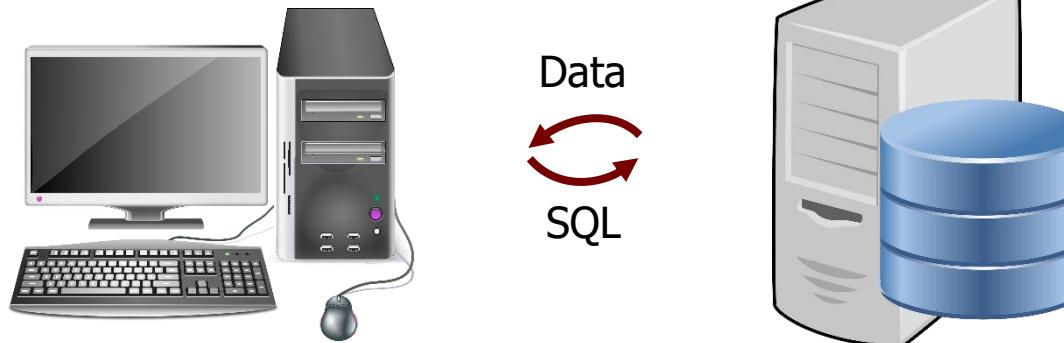
The Course Environment

Using SQL Developer

Chapter Summary

The Role of SQL

- SQL is a common interface between client and database server
- SQL is the interface between the application program and the database
 - SQL stands for Structured Query Language
 - But SQL is really a data sublanguage, which is more like an access method than a complete programming language



The Role of SQL (continued)

- Application programs may be:
 - 3GL programs
 - 4GL or application generator programs
 - Report generator programs
 - End-user point-and-click programs
 - Spreadsheets
 - Any frontend tool with SQL interface capability
 - Stored PL/SQL procedures
- Your ability to produce real-world programs will depend on your ability to write SQL statements

Result-Oriented

- SQL is a result-oriented language
 - Specify the desired result rather than step-by-step instructions of what to do

Example:

- If we have this table:
- Then this query against the Scott schema:

```
SELECT deptno, dname  
FROM   dept  
WHERE  loc = 'DALLAS';
```



DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

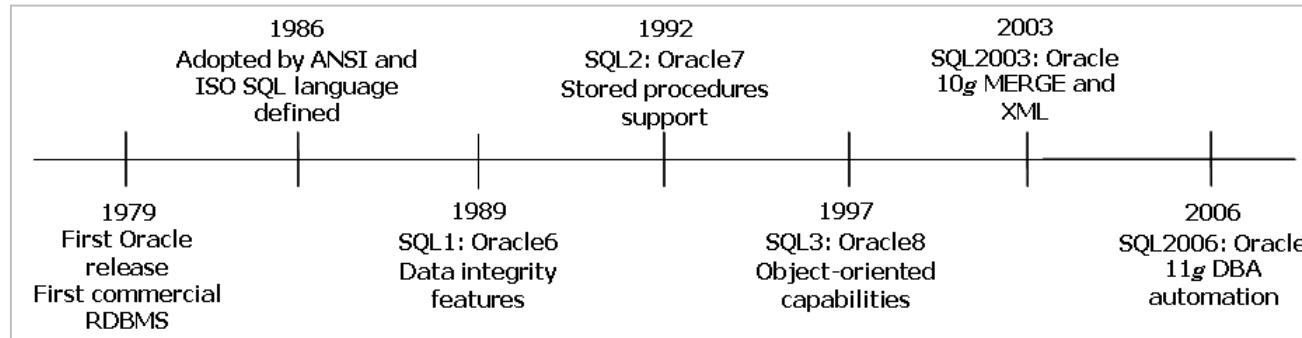
- Will produce this result:



DEPTNO	DNAME
20	RESEARCH

SQL Standard

- SQL is the standard language for relational databases
 - Unfortunately, different products implement commands differently
 - This course will adhere to the standards where possible
 - Some topics will be non-standard but will be indicated as such
- SQL functionality has evolved significantly over the years



Chapter Concepts

What Is SQL?

Designing a Database

The Course Environment

Using SQL Developer

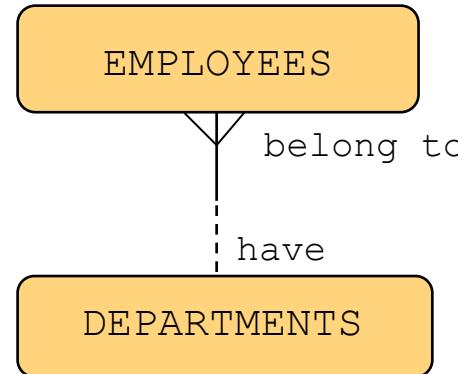
Chapter Summary

Logical Data Model

- Provides a level of abstraction from physical database design by representing data in terms of “logical” or business entities and the relationships between them
- Represents business information and rules
- Provides the input to physical database design
- Comprised of four critical elements
 - Entities
 - Attributes
 - Relationships
 - Candidate keys

Entity

- An object of importance
 - A uniquely identifiable person, place, thing, action, concept, object, or event about which information needs to be known or held
- Represented as a soft box
- Example:



Attribute

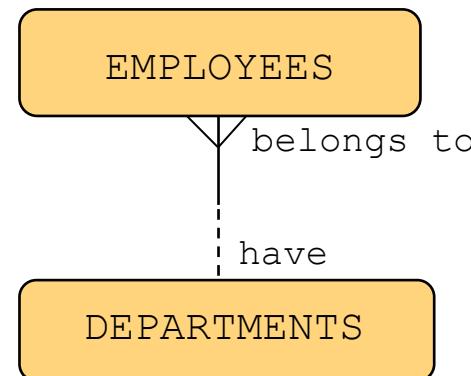
- A fact that is a nondecomposable unit of information about an entity
 - Qualify, identify, classify, quantify, or express the state of an entity
- Example:
 - `employee_id`, `last_name`, and `first_name` are attributes of the `employees` entity
- Further defined by indicating whether or not it is mandatory
 - That is, if it must exist for every occurrence of an entity
- Example:
 - `employee_id` is mandatory, whereas `phone_number` is not required

Attribute Datatype

- Each attribute is further qualified by a datatype
- Common datatypes are NUMBER, CHAR, and DATE
 - NUMBER represents numerical values, CHAR represents character strings, and DATE represents dates

Relationship

- Association between two entities
- Defined by a verb or a preposition connecting two entities
- Both ends must be named
- Example:



Relationship Cardinality

- Cardinality defines the expected number of related occurrences for each entity
- Most common cardinality is one to many (1:M)
- Example:
 - Department may have many employees, while each employee must represent one and only one department
- Indicates the parent entity (at the “one” end) and the child entity (at the “many” end)
- Example:
 - Departments is the parent and employees is the child

Relationship Optionality

- Defines coexistence of the two entities
- Defined at both ends of the relationship to indicate if a parent can exist without a child and if the child can exist without a parent
- Example:
 - Department may or may not have employees, whereas an employee must belong to a department

Reading Relationships

Relationships are read in both directions

- Cardinality and optionality are both included

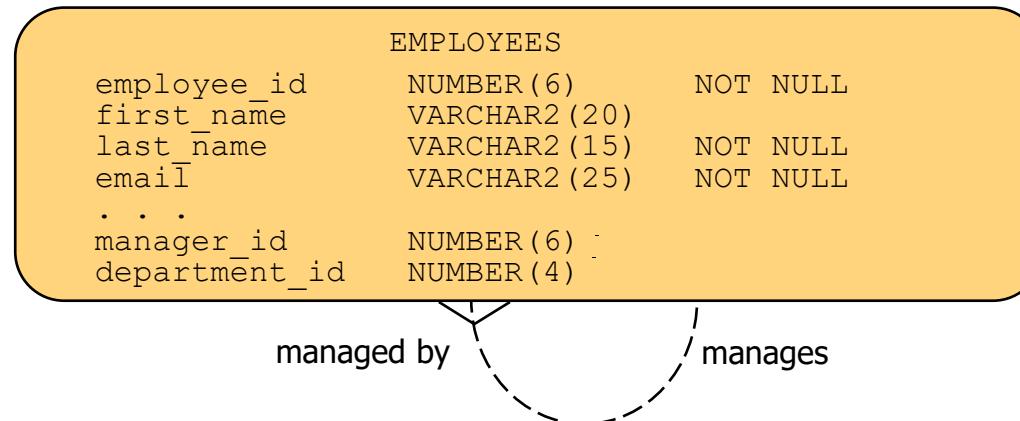
EACH	Entity1	MAY MUST	Relationship	ONE AND ONLY ONE ONE OR MORE	Entity2
------	---------	-------------	--------------	---------------------------------	---------

Examples:

- Each department **may** have **one or more** employees
- Each employee **must** belong to **one and only one** department

Recursive Relationship

- A relationship from an entity onto itself
- Captures a hierarchical structure, such as a reporting tree in an organization
- For example, a recursive relationship captures the fact that each employee may be managed by another employee
 - Each employee may manage one or more employees
 - Each employee may be managed by one and only one employee



Candidate Key

- An attribute or a minimal set of attributes that uniquely identify a specific row
- Examples:
 - `employee_id` uniquely identifies a employee
 - A combination of `first_name`, `last_name`, and `phone_number` uniquely identify an employee

Transforming a Logical Data Model to a Database Design

- Concepts in the logical model are mapped to database structures

Logical	Physical
Entity	Table
Attribute	Column
Candidate key	Primary or unique key
Relationship	Foreign key

- Tables and columns are usually a simple mapping from entities and attributes

Primary and Unique Keys

- A *unique key* has the same definition as a candidate key: a column or a minimum set of columns that uniquely identifies a specific row
- A *primary key* has the same definition as a unique key but with two further restrictions
 - It must be composed of mandatory columns
 - Only one primary key is allowed for each table
- Once the primary key is selected from a valid list of candidate keys, the remaining candidate keys are mapped to unique keys
- Example:
 - `employee_id` will be selected as a primary key because it is mandatory
 - The combination of `first_name`, `last_name`, and `phone_number` will become a unique key

Foreign Key

- By definition, a relationship copies candidate key columns of a parent table to a child table
- *Foreign key* enforces this relationship using two rules
 - Values in the relationship columns of the child table exist in the parent table
 - Cannot change values in the parent table that are referenced in the child table
- Example:
 - Foreign key corresponding to the department – employee relationship has these rules
 - The value of `department_id` in the `employee` table must previously exist in the `departments` table
 - The value of `department_id` in the `departments` table cannot be modified if it is used in the `employees` table

Chapter Concepts

What Is SQL?

Designing a Database

The Course Environment

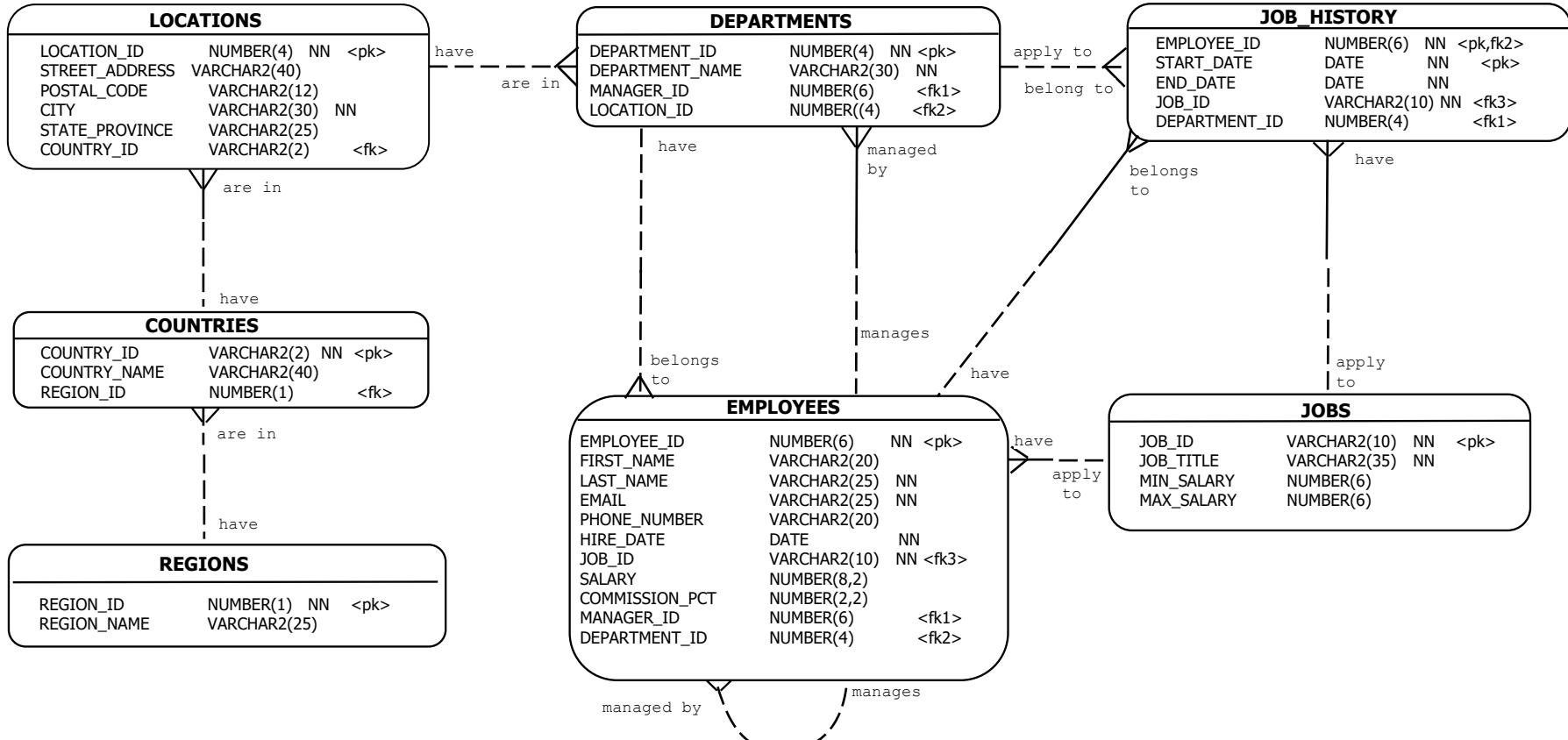
Using SQL Developer

Chapter Summary

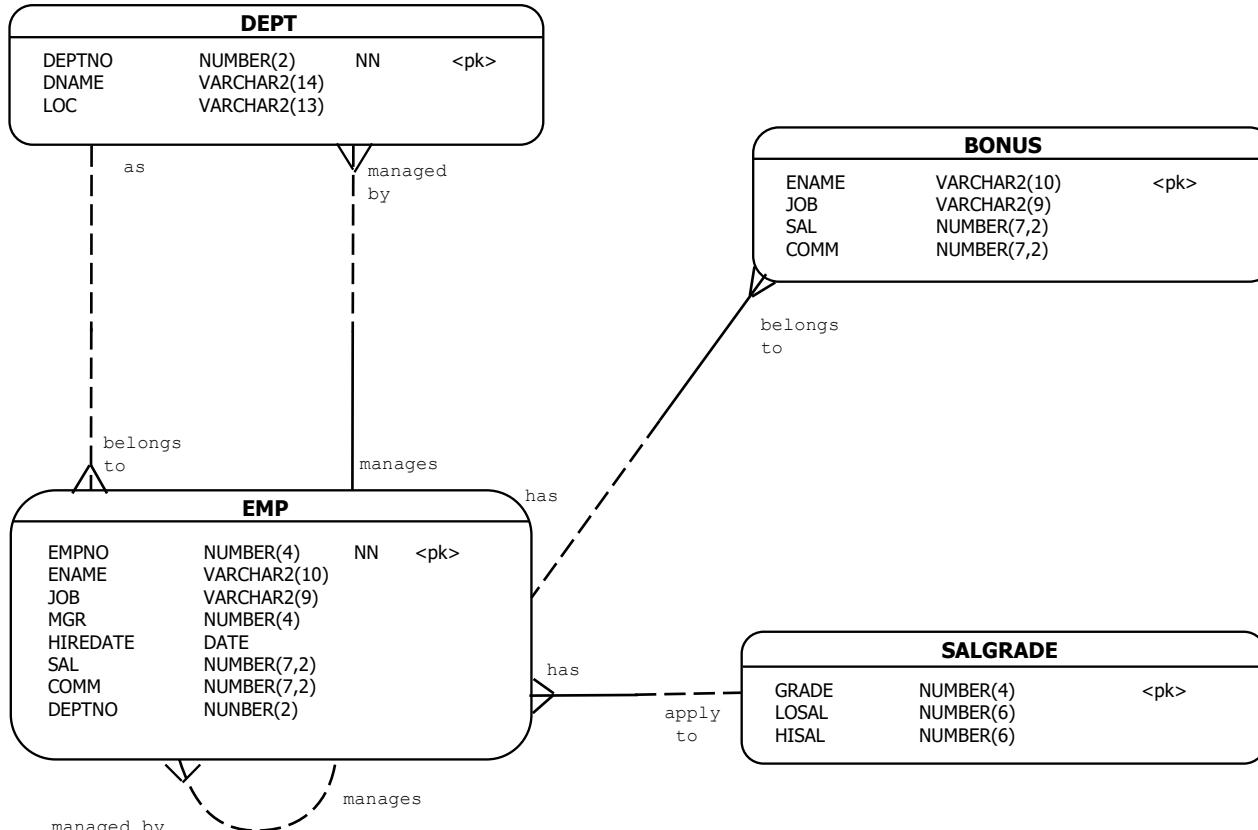
Course Exercises

- The exercises in this course all use standard Oracle user accounts
- When an Oracle Database is created, Oracle optionally embeds several users with populated schemas
 - These can then be used for training and demonstrating various features
- For experience in using multiple accounts, two of these user accounts (`HR` and `SCOTT`) have been selected for use
 - In each exercise, be careful to connect to the appropriate account
- The implication is that the students of this course can then go back and practice the lab exercises at any time without needing special setups from ROI

ER Diagram—HR Account



ER Diagram—SCOTT Account



Conventions for Command Syntax

- The following command syntax is used in this course:

Feature	Example	Explanation
Uppercase	CREATE	Reserved word; enter exactly as spelled
Lowercase	column_name	Substitute an appropriate value
Three periods	role_name,...,role_name	Items may be repeated any number of times
Square brackets	[NOT NULL]	Optional item
Vertical bar	ON OFF	Alternative item; use one or the other

Chapter Concepts

What Is SQL?

Designing a Database

The Course Environment

Using SQL Developer

Chapter Summary

SQL Developer

- Oracle SQL Developer is a graphical tool used to work with a database
 - Simplifies basic tasks for DBAs and developers
 - Released in 2006
- Developed in Java
 - Runs on Windows, Linux, and Mac OS X
- Supports Oracle 9*i* and later
- Key concepts
 - Connections
 - Object Navigator
 - SQL Worksheet

Connections

- Each connection is configured for a single Oracle user
 - Uses standard Oracle database authentication
- Can also connect to third party databases
 - Access, SQL Server, MySQL
- All connections are listed in the Connections window
 - Drill down each connection to view the objects to which the user has access
- Create a new connection using the icon (+) at the top of the Connections window
 - Requires information about the server, such as user, password, server name

Object Navigator and Details

- Expanding a Connection node exposes the Object Navigator
 - Automatically connects if not connected yet
- First level under the connection is a list of object types such as tables, views, and indexes
- Next level contains a list of objects
 - Example: table names (departments, employees, ...)
- When an object is selected, specific information is displayed
 - Information varies by object type
 - Selected by clicking an object in the Object Navigator
- Tables have the following commonly used tabs:
 - Columns—displays the structure of the table (columns, datatypes, etc.)
 - Data—displays the data from the table
 - SQL—includes the SQL to create the object

SQL Worksheet

- SQL Worksheet allows you to enter SQL and PL/SQL statements
 - Also supports some SQL*Plus commands
- Top window is a SQL statement editor
 - Supports both DML and DDL statements
 - Examples: creating tables, inserting data, selecting data
- Bottom windows display results
- Key components
 - Editor
 - Results window
 - Script Output window

Editor

- Used for writing SQL statements and executing scripts
 - Oracle keywords are highlighted automatically
 - Supports standard file operations such as open, save, and print
 - The Eraser icon clears the contents of the Editor
- To pull in a column or a table, drag it from the Object Navigator
 - For tables, a SQL statement is created automatically
- To format the statement, right-click in the Editor and select Format SQL
- To recall a previous command, click SQL History icon
 - History is maintained even if you close SQL Developer

Results Window

- Displays output from a single `SELECT` statement
 - Execute using Execute Statement icon or `<F9>` function key
- If multiple statements exist in the editor, only the one where the cursor is located will be executed
 - The line where the cursor exists is highlighted
 - Each statement must end with a semicolon, otherwise an error is displayed
- To sort data by one column, double-click the column heading
 - Once for ascending, a second time for descending
 - To sort by multiple columns, use an `ORDER BY` clause
- Right-click in the Results window to use the following features:
 - Auto Fit to format column widths
 - Count Rows to get the total number of records returned
 - Single Record View to view a single record at a time

Script Output Window

- Displays results of all commands in the editor
 - Execute using Run Script icon or <F5> function key
- Each statement in the editor is executed one after another
- For each statement, all results are displayed one after another
 - In contrast, the Results Window displays results of only one statement and only 50 rows at a time
 - Number of rows can be changed in Preferences Worksheet Parameters
- Use icons at the top of the Script Output Window to:
 - Clear the contents of the window
 - Save the contents to a file
 - Print the contents

Exporting Data

- Data can be exported from the Results Window and the Table Data Tab
 - Right-click in the Results Window and select Export Data
- Export Data pop-up provides the following features:
 - Output to a File or the Clipboard
 - Choose a file format such as TEXT, CSV, XML, HTML, XLS
 - Select the columns to include:
 - Default is ALL
 - Enter a WHERE clause to restrict the results
- The Apply button generates the file



HANDS-ON
EXERCISE

20 min

Exercise 1.1: Using SQL Developer

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is SQL?

Designing a Database

The Course Environment

Using SQL Developer

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The role of SQL
- Concepts of data modeling
- The course environment
- SQL Developer
- The Fidelity Development Environment

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 2: SQL Query Syntax

Chapter Overview

In this chapter, we will explore:

- Building basic SELECT statements
- Using the WHERE clause and the comparison operators
- Sorting the result set using the ORDER BY clause

Chapter Concepts

Building Basic SELECT

The WHERE Clause

The ORDER BY Clause

Chapter Summary

The Nature of SQL Statements

- SELECT, like all SQL statements, is a non-procedural, descriptive data access command
 - We describe WHAT we want, not HOW to do it
- The description is implemented by key words, followed by clauses that modify the key word
 - The clauses can have one or more entries
 - Not all key words must appear in the statement
 - Only SELECT and FROM are always required
 - Even if the statement does not need data from a table

Structure of the SELECT Statement

```
SELECT
    column or expression, column or expression ...
FROM
    table
WHERE
    condition 1 AND/OR condition 2 ...
ORDER BY
    column or expression or column alias or position, ...
```

The SELECT List

- The data to be returned is defined in the `SELECT` list
- Data elements are comma delimited
- The most common data elements are columns from some table
 - Anything that is in scope can be `SELECTed`
 - Any column in a table in the `FROM` clause
 - Literals
 - Expressions
 - Function calls returning data
- By default, Oracle will use the name of the column for the heading
 - Frontend tools format the width of the data based upon the definition of the column stored in the Data Dictionary

SELECT List Examples

- A column, an expression, or a literal can be SELECTed

```
SELECT last_name, salary, salary * 12, 'Wow', 1/8 FROM employees;
```

LAST_NAME	SALARY	SALARY*12	'WO'	1/8
King	24000	288000	Wow	.125
Kochhar	17000	204000	Wow	.125
De Haan	17000	204000	Wow	.125

- A literal can be selected from any table, but will be returned in a multiple number of rows

```
SELECT 1/8 FROM employees;
```

```
1/8
-----
.125
.125 ...
107 rows selected.
```

The Dummy Table, DUAL

- Oracle provides a table named `DUAL` to provide a workaround for the ANSI requirement that every `SELECT` statement *must* have a `FROM` clause
 - Useful when the information is not in a particular table

```
SELECT 1/8 FROM DUAL;
```

```
1/8
```

```
-----
```

```
.125
```

SELECT All Columns from a Table

- First, determine the columns using the DESCRIBE command:

```
desc jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

- Then, issue the SELECT statement

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000

- Notice the presentation sequence of the columns

Column Alias

- A column alias can also be used to override the default heading name
 - Syntax:
 - column AS column_alias, ...
 - AS is an optional key word
 - column column_alias , ...
 - Prefer using AS since it makes your intentions clear and avoids missing commas
 - The column alias can be a string with no spaces, or be bound within double quotes
 - Double quotes will respect the case of the string

```
SELECT job_title AS position, job_title AS "Position Title" FROM jobs;
```

POSITION

Position Title

President

President

Administration Vice President

Administration Vice President

NULLS

- The data value can be NULL
 - Meaning, a value has not been assigned
- If the value of a column is NULL:
 - Then it will be displayed as blank in script output
 - Or as (null) in the results window tabular view

```
SELECT city, state_province, country_id FROM locations;
```

	CITY	STATE_PROVINCE	COUNTRY_ID
1	Roma	(null)	IT
2	Venice	(null)	IT
3	Tokyo	Tokyo Prefecture	JP
4	Hiroshima	(null)	JP
5	Southlake	Texas	US
6	South San Francisco	California	US

ALL OR DISTINCT

- Relational theory mandates that all tuples (column values) in a set be unique
 - Not the default with SQL
- The implied set is defined by ALL
 - SELECT ALL ...
- If we only want the unique rows, we add DISTINCT (or UNIQUE) to the SELECT List
 - SELECT DISTINCT ...
 - Distinct applies to the *entire* SELECT list
 - Not just the column it appears in front of

ALL or DISTINCT Example

```
SELECT country_id  
FROM locations;
```

CO
--
IT
IT
JP
JP
US
US
US
US
US
CA
CA
CN

23 rows selected.

```
SELECT DISTINCT country_id  
FROM locations;
```

CO
--
AU
BR
CA
CH
CN
DE
IN
IT
JP
MX
NL
SG
UK
US

14 rows selected.

DISTINCT What?

- Remember that the DISTINCT applies to the entire select list

```
SELECT DISTINCT country_id, city FROM locations;
```

CO CITY

```
-- -----  
AU Sydney  
BR Sao Paulo  
CA Toronto  
CA Whitehorse  
CH Bern  
CH Geneva  
CN Beijing  
DE Munich  
IN Bombay  
IT Roma  
IT Venice
```

Chapter Concepts

Building Basic SELECT

The WHERE Clause

The ORDER BY Clause

Chapter Summary

The WHERE Clause

- Filters rows of data out of the result set
 - Predicated upon condition(s) testing True, False, or NULL
 - NULL, in a condition, always evaluates to False
- Example:
 - Restrict information about JOBS to those with a minimum salary of exactly 4000

```
SELECT *
FROM jobs
WHERE min_salary = 4000;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000

Constructing the Value to Be Tested: Strings

- String literals, sometimes referred to as character literals, are placed in single quote marks:
 - 'This is a string literal'
 - Any valid character can be part of a string literal
 - Including the single quote, which is escaped by another single quote
 - 'This is Oracle''s character set'

```
SELECT *
FROM jobs
WHERE job_title = 'Marketing Manager';
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
MK_MAN	Marketing Manager	9000	15000

Case Sensitivity

- String literals ARE case sensitive
 - 'A' and 'a' are not the same

```
SELECT *  
FROM jobs  
WHERE job_title = 'MARKETING MANAGER';  
  
no rows selected
```

- Handling case sensitivity
 - Some shops have the standard that ALL strings stored in the database must be in UPPERcase
 - May not be feasible
 - Can also be handled in SQL statement with functions (covered later)

Comparison Operators

- The test does not always have to be equal to (=)
- Other comparison operators include:
 - Not equal to specified as <> OR !=
 - Greater than > , less than <
 - Greater than or equal to >= , less than or equal to <=

```
SELECT *
FROM jobs
WHERE min_salary <> 4000;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
...			

More Comparison Operators: BETWEEN

- BETWEEN is used to describe a range of values inclusive of the end values
 - BETWEEN a AND b
 - The operator includes both end points
 - Both value a and value b will test TRUE

```
SELECT *
FROM jobs
WHERE min_salary BETWEEN 3000 AND 4000;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_ASST	Administration Assistant	3000	6000
IT_PROG	Programmer	4000	10000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000

- NOT BETWEEN is the logical opposite
 - The above four rows would NOT be included in the result set

More Comparison Operators: IN

- IN tests to determine if it is in a list of values

- IN (a, b, c)

```
SELECT *
FROM jobs
WHERE min_salary IN (3000, 4000);
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_ASST	Administration Assistant	3000	6000
IT_PROG	Programmer	4000	10000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000

- NOT IN is the logical opposite
 - NOT IN (x, y, z)

More Comparison Operators: LIKE

- LIKE tests a string for some sequence of characters
 - Character strings are enclosed in single 'quotes'
 - Remember that string literals are case sensitive
- Two wild card characters can be used to test for unspecified values
 - % means any value and zero or more characters
 - _ means any single character
- NOT LIKE is the logical opposite
- For example, list the names of all employees whose last name begins with P

```
SELECT first_name, last_name  
FROM employees  
WHERE last_name LIKE 'P%';
```

FIRST_NAME	LAST_NAME
Karen	Partners
Valli	Pataballa
Joshua	Patel
Randall	Perkins
Hazel	Philtanker
Luis	Popp

Testing for NULL

- **NULL is never equal (or not equal) to anything**
 - **NULL is never less than or greater than any value**
 - **NULL is never equal to or not equal to itself!**
 - Testing against **NULL is always false**
- **Testing to be = NULL is legal syntax**
 - But no rows will ever be selected

```
SELECT * FROM jobs WHERE min_salary = NULL;
```

no rows selected

```
SELECT * FROM jobs WHERE min_salary <> NULL;
```

no rows selected

Testing for NULL (continued)

- Must use the comparison operator `IS NULL`

```
SELECT city, state_province, country_id  
FROM locations  
WHERE state_province IS NULL;
```

CITY	STATE_PROVINCE	COUNTRY_ID
1 Roma	(null)	IT
2 Venice	(null)	IT
3 Hiroshima	(null)	JP
4 Beijing	(null)	CN
5 Singapore	(null)	SG
6 London	(null)	UK

- `IS NOT NULL` is the logical opposite

More than One Condition

- Multiple conditions (Boolean logic) can be constructed
 - Can specify an unlimited number of conditions
 - As long as the relationship between them is stated
 - Need to specify the logical operator: AND, OR, NOT

```
SELECT *
FROM jobs
WHERE min_salary = 3000 OR min_salary = 4000;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_ASST	Administration Assistant	3000	6000
IT_PROG	Programmer	4000	10000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000

Multiple Conditions: AND

- AND means both must be true

```
SELECT *
FROM jobs
WHERE min_salary = 4000 AND max_salary < 10000;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000

- Suppose we wanted all the rows other than these?

NOT-ing the Evaluation

- It is legal to NOT the paired conditional logic
 - Similar to applying NOT to a single condition
- Syntax: group the conditions with parentheses and NOT the group

```
SELECT *
FROM jobs
WHERE NOT (min_salary = 4000 AND max_salary < 10000);
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000

- Notice that this describes the opposite set
 - All the rows not included before

Specifying the Sequencing of Conditional Evaluation

- By default, Oracle will follow this precedence:
 - NOTs evaluated first, then ANDs, then ORs
 - Use parentheses to override the default evaluation order
- Since few people remember the order, favor parentheses for clarity
 - Also improves readability
 - And makes code more robust when being modified
- Question: Do the following statements describe the same data?

```
SELECT * FROM jobs
WHERE job_title = 'Marketing Manager'
OR min_salary = 4000 AND max_salary < 10000 ;
```

```
SELECT * FROM jobs
WHERE (job_title = 'Marketing Manager'
OR min_salary = 4000) AND max_salary < 10000 ;
```

Chapter Concepts

Building Basic SELECT

The WHERE Clause

The ORDER BY Clause

Chapter Summary

ORDER BY

■ ORDER BY is the next clause of the SELECT statement

- Its objective is to sort the result set
- Does not change any of the data being returned
- Just the output sequence
 - Otherwise, the data is in a “heap”
- The rows in the order of Oracle’s most efficient retrieval method

Syntax of ORDER BY

- Ordering can be specified by column name, column expression, column alias, select list position
 - The first is the primary sort, the second is sorted within the primary
 - The default sequence is `ASC`ending
 - Usually not specified
 - Sort order can be `DESC`ending
 - `NULL`s sorts high
 - The column being sorted on does not have to be in the `SELECT` list
 - Any column in any table in the query can be referenced

Which Job Title Do I NOT Want?

```
SELECT job_title, max_salary  
FROM jobs  
ORDER BY max_salary;
```

JOB_TITLE	MAX_SALARY
Stock Clerk	5000
Purchasing Clerk	5500
Shipping Clerk	5500
Administration Assistant	6000
Stock Manager	8500
Accountant	9000
Public Accountant	9000
Marketing Representative	9000
Human Resources Representative	9000
Programmer	10000
Public Relations Representative	10500
Sales Representative	12000
Purchasing Manager	15000
Marketing Manager	15000
Finance Manager	16000
Accounting Manager	16000
Sales Manager	20000
Administration Vice President	30000
President	40000

19 rows selected.

Which Job Title Do I Want?

```
SELECT job_title, max_salary  
FROM jobs  
ORDER BY max_salary DESC;
```

JOB_TITLE	MAX_SALARY
President	40000
Administration Vice President	30000
Sales Manager	20000
Finance Manager	16000
Accounting Manager	16000
Purchasing Manager	15000
Marketing Manager	15000
Sales Representative	12000
Public Relations Representative	10500
Programmer	10000
Accountant	9000
Public Accountant	9000
Human Resources Representative	9000
Marketing Representative	9000
Stock Manager	8500
Administration Assistant	6000
Purchasing Clerk	5500
Shipping Clerk	5500
Stock Clerk	5000

19 rows selected.

Ordering on an Expression

- The expression could be repeated ...

```
SELECT job_title, max_salary / 12 AS "Monthly Salary"  
FROM jobs  
WHERE max_salary / 12 > 1000  
ORDER BY max_salary / 12 DESC, job_title;
```

JOB_TITLE	Monthly Salary
President	3333.33333
Administration Vice President	2500
Sales Manager	1666.66667
Accounting Manager	1333.33333
Finance Manager	1333.33333
Marketing Manager	1250
Purchasing Manager	1250

7 rows selected.

Ordering on a Column Alias

- This is legal because the ORDER BY happens after the result set has been determined
 - And the column aliases have been applied to the set

```
SELECT job_title, max_salary / 12 AS "Monthly Salary"  
FROM jobs  
WHERE max_salary / 12 > 1000  
ORDER BY "Monthly Salary" DESC, job_title;
```

JOB_TITLE	Monthly Salary
President	3333.33333
Administration Vice President	2500
Sales Manager	1666.66667
Accounting Manager	1333.33333
Finance Manager	1333.33333
Marketing Manager	1250
Purchasing Manager	1250

7 rows selected.

Ordering by SELECT List Position

- This is still legal, but it is not a good practice
 - Useful for ad hoc statements

```
SELECT job_title, max_salary / 12 AS "Monthly Salary"  
FROM jobs  
WHERE max_salary / 12 > 1000  
ORDER BY 2 DESC, job_title;
```

JOB_TITLE	Monthly Salary
President	3333.33333
Administration Vice President	2500
Sales Manager	1666.66667
Accounting Manager	1333.33333
Finance Manager	1333.33333
Marketing Manager	1250
Purchasing Manager	1250

7 rows selected.

Positioning NULLS: High by Default

- Given the following set of data:

By default, NULLs sort high

```
SELECT department_name,  
       manager_id  
  FROM departments  
 ORDER BY manager_id DESC;
```

13	Corporate Tax	(null)
14	Payroll	(null)
15	Shareholder Services	(null)
16	Benefits	(null)
17	Accounting	205
18	Public Relations	204
19	Human Resources	203

```
SELECT department_name,  
       manager_id  
  FROM departments  
 ORDER BY manager_id;
```

NAME	MANAGER_ID
Finance	100
Purchasing	103
Shipping	108
Sales	114
Administration	121
Marketing	145
Human Resources	200
Public Relations	201
Accounting	203
IT Helpdesk	204
Government Sales	205
Retail Sales	(null)

Forcing the Placement of NULLS

- The options on the ORDER BY clause include:

- NULLS FIRST and NULLS LAST
- This forces the NULLS to (positionally) be on the top or bottom
 - Without regard to whether the sort order is ASC or DESC

```
SELECT department_name, manager_id  
FROM departments  
ORDER BY manager_id DESC NULLS FIRST;
```

13	Corporate Tax	(null)
14	Payroll	(null)
15	Shareholder Services	(null)
16	Benefits	(null)
17	Accounting	205
18	Public Relations	204
19	Human Resources	203

```
SELECT department_name, manager_id  
FROM departments  
ORDER BY manager_id DESC NULLS LAST;
```

8	Purchasing	114
9	Finance	108
10	IT	103
11	Executive	100
12	IT Helpdesk	(null)
13	Government Sales	(null)
14	Retail Sales	(null)



HANDS-ON
EXERCISE

60 min

Exercise 2.1: Selecting Data

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Building Basic SELECT

The WHERE Clause

The ORDER BY Clause

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Building basic SELECT statements
- Using the WHERE clause and the comparison operators
- Sorting the result set using the ORDER BY clause



Technology Immersion Program

Oracle Database: A Comprehensive Introduction

Chapter 3: SQL Scalar Functions

Chapter Objectives

In this chapter, we will explore:

- Common datatypes
- Simple SQL functions
 - Definition
 - Classes of functions
 - Common Single Row (Scalar) Functions

Chapter Concepts

Basic Server Datatypes

Introduction to Functions

Scalar Functions

Chapter Summary

NUMBER Datatype

■ Used to store fixed or floating-point numbers

■ Syntax:

NUMBER [(precision, scale)]

■ Precision

- Total number of significant digits
- Optional, defaults to the maximum (38 digits)

■ Scale

- Number of digits after the decimal point
- Can range from -84 to 127
- Optional, defaults to zero

■ Examples:

- NUMBER
 - 38 total digits (before or after the decimal point)
- NUMBER (2)
 - Two digits before the decimal point and zero digits after
- NUMBER (3, 2)
 - One digit before the decimal point and two digits after
- NUMBER (*, 2)
 - 38 digits of total precision with two digits after the decimal point

CHAR Datatypes

- CHAR is used to store fixed-length character data

- Syntax:

CHAR [(length)]

- Length

- Maximum 2,000 bytes
 - Optional, defaults to 1
 - Values are padded with blanks to the maximum length

- Example:

- CHAR stores one character
 - CHAR(10) stores 10 characters for a value of any length
 - Storing 'MIKE' in this datatype would result in MIKE and six blanks

VARCHAR2 Datatype

- Used to store variable-length character data
 - Syntax:
VARCHAR2 (length)
- Length
 - Maximum 4,000 bytes
 - Since Oracle 12c, there is a database option to allow strings as long as 32767 bytes
 - Mandatory
 - Values are not padded; exactly the length of the string is stored
- Example:
 - VARCHAR2 (10) stores up to 10 characters based on the actual string
 - Storing 'MIKE' in this datatype would not store extra characters

DATE Datatype

- Used to store date and time to the precision of seconds
 - Syntax:
DATE
- Stored internally as an ordered set of seven bytes, representing century, year, month, day, hour, minute, second
 - All DATEs contain a date and a time (differs from the standard)
 - If the time is not set, it defaults to midnight
 - If the date is not set, it defaults to the first day of the current month
- Can add and subtract dates
 - `start_date + 1` is one day after start date
 - `end_date - start_date` is the number of days in this period

Date Literals

- Oracle will interpret certain character literals as dates when needed
 - Relies on the default format mask set by `NLS_DATE_FORMAT`
 - Defaults to '`DD-MON-YY`', e.g., '`21-JAN-01`'
 - This course is set to '`DD-MON-YYYY`', e.g., '`21-JAN-2001`'
- ***Do not rely on this***
 - You cannot always control the format
 - It is fine for testing, but not production code
- Either use the ANSI standard date format
 - `DATE 'YYYY-MM-DD'`, e.g., `DATE '2001-01-01'`
 - Only useful for setting dates, does not support times
- Or use `TO_DATE(char_literal, format)`
 - E.g., `TO_DATE('98-DEC-25 17:30', 'YY-MON-DD HH24:MI')`

https://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements003.htm#BABGIGCJ

Date and Time Format Models

- Commonly used format models for DATE and TIMESTAMP datatypes

Format Model	Meaning
YYYY	Four-digit year
YY	Two-digit year
MON	Three-character name of month
MM	Two-digit month
DD	Two-digit day
HH	Two-digit hour of day (1–12)
AM	Two-character meridian indicator
HH24	Two-digit hour of day (0–23)
MI	Two-digit minute (0–59)
SS	Two-digit seconds (0–59)
FF	Fractional seconds (1–9 digits)

TIMESTAMP Datatype

- Extension of the DATE datatype that supports fractional seconds

- Syntax:

```
TIMESTAMP [ (precision) ]
```

- Precision specifies the number of digits in the fractional part of seconds that will be stored and displayed
 - Can be a number in the range 0 to 9 (default is 6 digits)
- Retrieved and updated based on date and time format models
 - Default format mask is 'DD-MON-YY HH.MI.SS.FF AM'
 - Changed by setting NLS_TIMESTAMP_FORMAT parameter
 - This course is set to 'DD-MON-YYYY HH24:MI:SS.FF'
 - Example of four-digit precision: '21-JAN-2001 20:12:10.0250'
 - ***Do not rely on this, use ANSI: TIMESTAMP '2001-01-21 20:12:10.0250'***

Implicit Datatype Conversion

From\to	CHAR	VARCHAR2	DATE	TIMESTAMP	NUMBER
CHAR		Yes	Yes	Yes	Yes
VARCHAR2	Yes		Yes	Yes	Yes
DATE	Yes	Yes		No	No
TIMESTAMP	Yes	Yes	No		No
NUMBER	Yes	Yes	No	No	

- Implicit conversion to dates and timestamp requires that the string be in the default date or timestamp format

Chapter Concepts

Basic Server Datatypes

Introduction to Functions

Scalar Functions

Chapter Summary

Functions

■ Manipulate data items and return a result

- Modify a value
- Combine values
- Change value formats
- Create new values

■ Syntax:

FUNCTION_NAME (parameter1, parameter2, ... parameterN)

- Some functions require no parameters

■ Most SQL functions are ANSI compliant

- There is a standard specification that vendors adhere to
- They will work the same with any RDBMS that is ANSI compliant

■ Oracle, like other vendors, supplies functions that are not ANSI

- Considered extensions to standard SQL

Classes of Functions

- Functions are classified according to nature of the data they are working on
- Single Row (or scalar) Functions
 - Single-row functions return a single result row for every row of a queried table or view
 - These are the type we will be discussing in this chapter
- Aggregate functions
 - Return a single result row based on groups of rows, rather than on single rows
 - Covered later

Single Row (Scalar) Functions: Types

- There are over 150 Single Row Functions
- We will look at the most commonly used functions
 - Numeric
 - Character or string
 - Date/Time
 - Analytical
 - Miscellaneous
- Others handle explicit conversion of datatypes

Chapter Concepts

Basic Server Datatypes

Introduction to Functions

Scalar Functions

Chapter Summary

Numeric Functions

- Accept numeric input and return numeric values
 - Most numeric functions that return NUMBER values that are accurate to 38 decimal digits
- The numeric functions are:

ABS
ACOS
ASIN
ATAN
ATAN2
BITAND
CEIL
COS
COSH

EXP
FLOOR
LN
LOG
MOD
NANVL
POWER
REMAINDER
ROUND (number)

SIGN
SIN
SINH
SQRT
TAN
TANH
TRUNC (number)
WIDTH_BUCKET

ROUND: Numeric

ROUND(the_numeric_value, the_degree_of_rounding)

- If the second parameter is not supplied, then 0 decimal positions is assumed
- If the second parameter is negative, rounds to the left of the decimal

```
SELECT 123.45
,ROUND(123.45)      AS A1 ,ROUND(123.45,1)   AS B1 ,ROUND(123.45,2)   AS C1
,ROUND(123.45,-1)  AS D1 ,ROUND(123.45,-2)  AS E1   FROM DUAL;
```

123.45	A1	B1	C1	D1	E1
123.45	123	123.5	123.45	120	100

```
SELECT 123.55
,ROUND(123.55)      AS A2 ,ROUND(123.55,1)   AS B2 ,ROUND(123.55,2)   AS C2
,ROUND(123.55,-1)  AS D2 ,ROUND(123.55,-2)  AS E2   FROM DUAL;
```

123.55	A2	B2	C2	D2	E2
123.55	124	123.6	123.55	120	100

TRUNC: Numeric

TRUNC(the_numeric_value, the_degree_of_rounding)

- If the second parameter is not supplied, then 0 decimal positions is assumed
- If the second parameter is negative, truncates to the left of the decimal

```
SELECT 123.55
      ,TRUNC(123.55)      AS A
      ,TRUNC(123.55,1)    AS B
      ,TRUNC(123.55,2)    AS C
      ,TRUNC(123.55,-1)   AS D
      ,TRUNC(123.55,-2)   AS E
FROM DUAL;
```

123.55

A

B

C

D

E

123.55

123.00

123.50

123.55

120

100

Character Functions: Returning Characters

■ Character functions that return character values

- Also referred to as string functions
- The length of the value returned by the function is limited by the maximum length of the datatype returned

■ The character functions that return character values are:

CHR
CONCAT or ||
INITCAP
LOWER
LPAD
LTRIM
NLS_INITCAP
NLS_LOWER

NLSSORT
NLS_UPPER
REGEXP_LIKE
REGEXP_REPLACE
REGEXP_SUBSTR
REPLACE
RPAD
RTRIM

SOUNDEX
SUBSTR
TRANSLATE
TREAT
TRIM
UPPER

Character Functions: Concatenating

- CONCAT (parameter1, parameter2) returns a single string
- The more useful construction is ||
 - Can string more than one parameter together
 - Oracle provides implicit datatype conversions
- Example:

```
SELECT first_name || ' ' || last_name || ' was hired on ' || hire_date  
FROM employees  
WHERE employee_id IN (163,164);  
  
FIRST_NAME || ' ' || LAST_NAME || 'WASHIREDON' || HIRE_DATE  
-----  
Danielle Greene was hired on 19-MAR-99  
Mattea Marvins was hired on 24-JAN-00
```

Character Functions: UPPER and LOWER

- Return the string in the specified case: `UPPER(value)`, `LOWER(value)`

```
SELECT LOWER(first_name) || ' ' || UPPER(last_name) || ' was hired on ' || hire_date  
FROM employees  
WHERE employee_id IN (163,164);
```

```
LOWER(FIRST_NAME) || ' ' || UPPER(LAST_NAME) || 'WASHIREDON' || HIRE_DATE
```

```
-----  
danielle GREENE was hired on 19-MAR-99
```

```
mattea MARVINS was hired on 24-JAN-00
```

- Useful when the case of the character columns is not known, or not consistent

```
SELECT hire_date FROM employees WHERE last_name = 'GREENE';
```

```
no rows selected
```

```
SELECT hire_date FROM employees WHERE UPPER(last_name) = 'GREENE';
```

```
HIRE_DATE
```

```
-----  
19-MAR-1999
```

Character Functions: SUBSTR

- The SUBSTR functions return a portion of string

- Syntax: SUBSTR(some_string, position, substring_length)

- Returns the string beginning at position and length of substring_length
 - substring_length defaults to the end of the string
 - If position is negative, then it is relative to the end of the string

```
SELECT country_name
      , SUBSTR(country_name,1,2)      AS A
      , SUBSTR(country_name,1)        AS B
      , SUBSTR(country_name,5,3)      AS C
      , SUBSTR(country_name,-10,3)    AS D
      , SUBSTR(country_name,-4)       AS E
  FROM countries
 WHERE country_id = 'CH';
```

COUNTRY_NAM	A	B	C	D	E	
Switzerland	Sw		Switzer	zer	wit	land

Character Functions: TRIM, LTRIM, RTRIM

- TRIM enables you to trim leading or trailing characters (or both) from a character string

■ Syntax:

```
TRIM( [ [LEADING | TRAILING | BOTH ] trim_character FROM] source)
```

- Removes consecutive characters matching `trim_character` from specified position

- BOTH is the default
- If you do not specify `trim_character`, then the default value is a blank space
- So, `TRIM(column_name)` will remove leading and trailing blank spaces
- If either parameter is `NULL`, then the function returns `NULL`
- TRIM is an ANSI standard function

- Older Oracle functions are RTRIM and LTRIM

- `LTRIM(source, trim_characters)`
- Can only trim from one side, but can trim more than one character
- To trim BOTH: `LTRIM(RTRIM(source, trim_characters), trim_characters)`

Character Functions: TRIM, LTRIM, RTRIM Examples

```
SELECT job_title
      , TRIM(BOTH      'M'    FROM job_title) AS A
      , TRIM(LEADING   'M'    FROM job_title) AS B
      , TRIM(TRAILING  'R'    FROM job_title) AS C
      , TRIM(TRAILING  'r'    FROM job_title) AS D
  FROM jobs
 WHERE job_id = 'MK_MAN';
```

JOB_TITLE	A	B	C	D
Marketing Manager	arketing Manager	arketing Manager	Marketing Manager	Marketing Manage

```
SELECT job_title
      , LTRIM(job_title, 'M')                      AS A
      , LTRIM(RTRIM(job_title, 'M'), 'M')           AS B
      , LTRIM(RTRIM(job_title, 'Mare'), 'Mare')     AS C
  FROM jobs
 WHERE job_id = 'MK_MAN';
```

JOB_TITLE	A	B	C
Marketing Manager	arketing Manager	arketing Manager	keting Manag

Character Functions: Returning a Number

- Character functions that return number values can take as their argument any character datatype
- The character functions that return number values are:

ASCII
INSTR
LENGTH
REGEXP_INSTR

Character Functions: INSTR

■ The INSTR functions search string for substring

■ Syntax:

```
INSTR(string, substring, position, occurrence)
```

- Returns an integer indicating the position of substring in string
 - Value is the position of the first character of substring in this occurrence
- position is the character position in string where the search begins
 - If negative, then INSTR counts and searches backward from the end of string
 - Default is 1
- occurrence indicates which occurrence of substring Oracle should search for
 - Must be positive, default is 1

Character Functions: INSTR Example

- Search for the strings of Marketing Manager

```
SELECT job_title
      , INSTR(job_title, 'M', 1, 1) AS A
      , INSTR(job_title, 'M', 1, 2) AS B
      , INSTR(job_title, 'M', 2, 1) AS C
      , INSTR(job_title, 'ark', 1, 1) AS D
      , INSTR(job_title, 'ark', 1, 2) AS E
   FROM jobs
 WHERE job_id = 'MK_MAN';
```

JOB_TITLE	A	B	C	D	E
Marketing Manager	1	11	11	2	0

- List all employee last names than contain an embedded blank

```
SELECT employee_id, first_name, last_name
  FROM employees
 WHERE INSTR(last_name, ' ') > 1;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME
102	Lex	De Haan

Character Functions: LENGTH

- The LENGTH functions return the length of strings

- Implicitly converts any datatype to a string if necessary
 - Can use to determine the “length” of a number
 - Or the “length” of a date datatype in default format

Syntax:

LENGTH(string)

- Example: find the length of the email address for employee ID 102

```
SELECT first_name, last_name, email, LENGTH(email)
FROM employees
WHERE employee_id = 102;
```

FIRST_NAME	LAST_NAME	EMAIL	LENGTH(EMAIL)
Lex	De Haan	LDEHAAN	7

TO_CHAR: A Formatting Function

■ The formatting function is TO_CHAR

- It takes two parameters
 - The data to be formatted
 - The format mask to be used

■ Example:

```
SELECT salary,  
       TO_CHAR(salary,'$99,999')           AS sal,  
       TO_CHAR(hire_date, 'Mon DD, YYYY') AS hired  
FROM employees;
```

SALARY	SAL	HIRED
-----	-----	-----
24000	\$24,000	Mar 10, 2002



HANDS-ON
EXERCISE

60 min

Exercise 3.1: Using Scalar Functions

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Basic Server Datatypes

Introduction to Functions

Scalar Functions

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Common datatypes
- Simple SQL functions
 - Definition
 - Classes of functions
 - Common Single Row (Scalar) Functions

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 4: SQL Joins

Chapter Objectives

In this chapter, we will explore:

- The use of `JOINS` to retrieve data from multiple tables
- The difference between `INNER`, `OUTER`, and `CROSS JOIN`
- Joining a table to itself
- Cartesian Join
- SQL-89 vs SQL-92 syntax

Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

The Need for Joins

- A well-designed Oracle database is normalized to 3rd Normal Form
 - 3NF means that the columns in each table describe:
 - The primary key
 - The WHOLE primary key
 - And NOTHING BUT the primary key
- The reason behind this concept is to ensure that data belongs with the other data it is associated with and dependent upon
 - This leads to having only one copy of each data element
 - This minimizes *update anomalies*
 - The application having to maintain multiple copies of the data element in more than one table

Three Normal Forms Are Most Often Practiced

- The physical implementation of most databases implement three normal forms to ensure data integrity
 - 1st Normal Form: declare a primary key and remove repeating groups
 - 2nd Normal Form: for composite keys, ensure functional dependency
 - 3rd Normal Form: functional dependency on only the primary key
- Additional Normal Forms:
 - 4th Normal Form: minimize the fields involved in a composite key
 - 5th Normal Form: removes all redundancies

Denormalized Data

- Employees work for a department
 - Departments have names
- Since requests for information about employees frequently want the department name on the listing, it is tempting to make the name a column in the employee table
- On the average, each department has around nine employees assigned to them
 - The implication is that the department name would have to be carried redundantly
 - Consuming more storage
 - Creating an *update anomaly*
 - What must we do if the department changes its name?

Normalized Data

- With 107 employees, this may not be an issue
 - Increasing table sizes compounds the problem
 - Having many denormalized data elements begins to exponentiate the problem
 - Eventually, an application person will forget (or not know) that the other copies of the data need to be maintained in synch
- This leads to a data integrity issue
 - Potentially, the most expensive problem
 - What is the impact on the business of a decision based upon inaccurate information?
- The solution is to design the database to 3NF
- In our case, this means that the SQL statement must *join* data from two or more tables to satisfy the request

Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

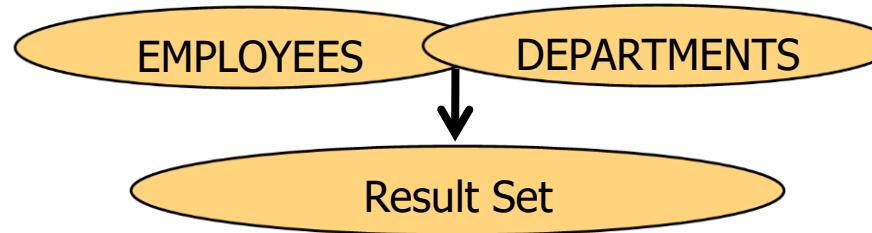
Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

Preferred Types of Attribute Joins

- In order of preference, there are multiple ways to do an attribute join
 - A pre-defined Foreign Key to Primary Key relationship
 - An indexed attribute in one table that links to an indexed attribute in another table
 - An index is created by a database administrator
 - Results in faster performance of the `SELECT` clause
 - Columns without keys or indices, but that match in terms of data type and data value
 - Retrieval will be slower on these types of joins



Join Example

- Consider the locations and departments tables

LOCATIONS

LOCATION_ID	CITY
1700	Seattle
2500	Oxford
2700	Munich

DEPARTMENTS

LOCATION_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1700	30	Purchasing
1700	90	Executive
1700	110	Accounting
1700	120	Treasury
2500	80	Sales
2700	70	Public Relations

- Each department belongs to a particular location
 - We would like to list the departments and include the names of their locations

LOCATION_ID	CITY	DEPARTMENT_ID	DEPARTMENT_NAME
2700	Munich	70	Public Relations
1700	Seattle	90	Executive
1700	Seattle	110	Accounting
1700	Seattle	120	Treasury
2500	Oxford	80	Sales
1700	Seattle	30	Purchasing
...			

Join Example (continued)

- The following query will produce the result on the previous slide:

```
SELECT locations.location_id  
      , locations.city  
      , departments.department_id  
      , departments.department_name  
FROM   departments  
JOIN    locations  
ON       departments.location_id = locations.location_id;
```

Explanation:

- A join returns columns from more than one table
 - In this case we need columns from two tables
- The `ON` condition specifies how the rows in the tables relate to one another
- The column names have prefixes to specify the table in which each column is located
 - Prefixes will be discussed in more detail later

Another Join Example

- For each employee, list the employee id, first name, last name, job id, and job title
 - job_title is a column in the jobs table
 - The other columns are located in the employees table

```
SELECT employees.employee_id  
      , employees.first_name  
      , employees.last_name  
      , employees.job_id  
      , jobs.job_title  
  FROM employees  
JOIN   jobs  
ON     employees.job_id = jobs.job_id;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
100	Steven	King	AD_PRES	President
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
109	Daniel	Faviet	FI_ACCOUNT	Accountant
108	Nancy	Greenberg	FI_MGR	Finance Manager
...				

Column Names

- A column name that occurs in more than one of the tables must have the table name as a prefix so as not to be ambiguous
- Prefixing all column names is strongly recommended, even when it is not strictly necessary
 - Improves readability
 - No risk of becoming ambiguous if:
 - New tables are added to the query
 - New columns are added to tables

FROM Clause and Table Alias

- FROM and JOIN clauses specify the tables being used in the statement
- A table can have an alias name
 - Also called *range variable* or *correlation name*
 - Like column aliases, the key word AS is specified in the standard
 - Most products do not require it
 - Oracle does *not allow* the key word AS with table alias names
 - For this reason, we will *omit* the AS with table alias names in the course examples
- The alias *replaces* the real table name *within* the query
 - The alias must be used as a prefix instead of the real table name
 - A table alias is useful to reduce typing and improve readability

Tables Alias Examples

```
SELECT last_name FROM employees;
```

LAST_NAME

```
-----  
Abel  
Ande  
Atkinson  
Austin
```

```
SELECT employees.last_name FROM employees;
```

LAST_NAME

```
-----  
Abel  
Ande  
Atkinson
```

```
SELECT e.last_name FROM employees e;
```

LAST_NAME

```
-----  
Abel  
Ande  
Atkinson  
Austin
```

```
SELECT employees.last_name FROM employees e;
```

*

ERROR at line 1:

ORA-00904: "EMPLOYEES"."LAST_NAME": invalid identifier

Re-write the Join with Table Aliases

Using aliases in the previous example:

– From:

```
SELECT employees.employee_id  
      , employees.first_name  
      , employees.last_name  
      , employees.job_id  
      , jobs.job_title  
FROM   employees  
JOIN   jobs  
ON     employees.job_id = jobs.job_id;
```

– To:

```
SELECT e.employee_id  
      , e.first_name  
      , e.last_name  
      , e.job_id  
      , j.job_title  
FROM   employees e  
JOIN   jobs j  
ON     e.job_id = j.job_id;
```

JOIN and WHERE

- It is possible to combine a WHERE condition with a JOIN
 - Example: Restrict the previous example to include only the job id of FI_ACCOUNT

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
  FROM employees e
 JOIN   jobs j
 ON     e.job_id = j.job_id
 WHERE   e.job_id = 'FI_ACCOUNT';
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
109	Daniel	Faviet	FI_ACCOUNT	Accountant
....				

JOIN and Sorting

- It is impossible to predict the sort order of a JOIN result
 - Unless ORDER BY is specified
 - Example: Sort the result of the previous query by job title and employee ID

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
  FROM employees e
 JOIN jobs j
 ON e.job_id = j.job_id
 WHERE e.job_id = 'FI_ACCOUNT'
 ORDER BY job_title, employee_id;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
109	Daniel	Faviet	FI_ACCOUNT	Accountant
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
....				

More than Two Tables

- There may be more than two tables in a JOIN
 - All tables must have a JOIN condition
 - The keywords JOIN and ON are repeated for each additional table
- Example: Include department name
 - Department name is a column in the departments table
 - The relationship between employees and departments is department ID

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
      , d.department_name
  FROM employees e
  JOIN jobs j
  ON e.job_id = j.job_id
  JOIN departments d
  ON e.department_id = d.department_id
 ORDER BY job_title, employee_id;
```

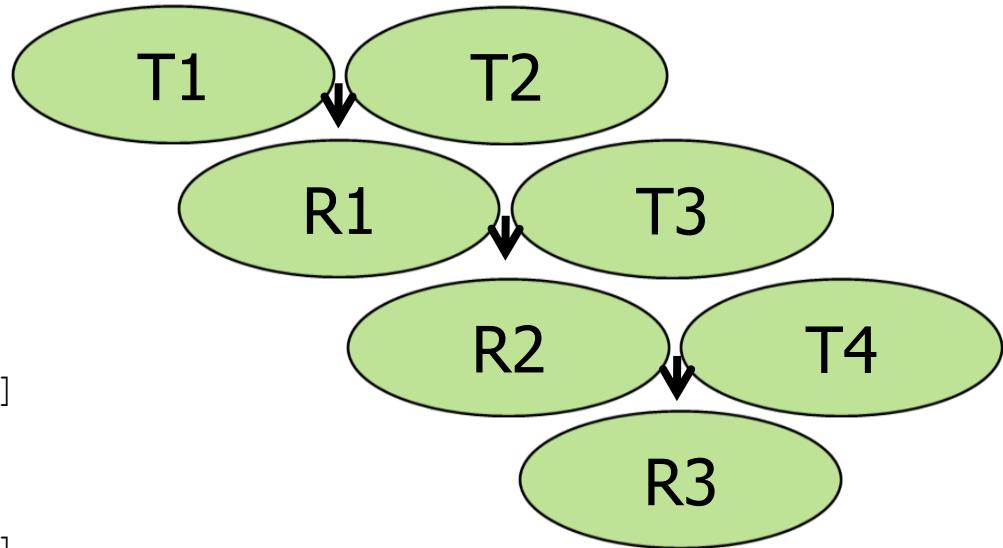
More than Two Tables (continued)

- Partial result of the query on the previous slide:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE	DEPARTMENT_NAME
167	Amit	Banda	SA_REP	Sales Rep	Sales
168	Lisa	Ozer	SA_REP	Sales Rep	Sales
197	Kevin	Feeney	SH_CLERK	Shipping Clerk	Shipping
198	Donald	OConnell	SH_CLERK	Shipping Clerk	Shipping
199	Douglas	Grant	SH_CLERK	Shipping Clerk	Shipping
....					

Visualizing Multiple Table JOINS

```
SELECT [columns]  
FROM TABLE1 T1  
JOIN TABLE2 T2  
    ON T1.[column]=T2.[column]  
JOIN TABLE3 T3  
    ON table.[column]=T3.[column]  
JOIN TABLE4 T4  
    ON table.[column]=T4.[column]
```



- Note: each JOIN statement joins TWO tables (or the results of a previous join to another table)

Other Join Conditions

- All the joins we have seen use the equals operator
 - Known as equijoins
- Joins may use any of the SQL conditional operators
 - Watch for multiple rows returned
 - These are usually less efficient than equijoins

```
SELECT e.first_name, e.last_name, e.salary, j.job_title, j.max_salary
FROM employees e
JOIN jobs j
ON e.salary BETWEEN j.min_salary AND j.max_salary
AND e.job_id != j.job_id
WHERE employee_id = 103;
```

FIRST_NAME	LAST_NAME	SALARY	JOB_TITLE	MAX_SALARY
Alexander	Hunold	8000	Accountant	9000
Alexander	Hunold	8000	Public Accountant	9000
...				



HANDS-ON
EXERCISE

60 min

Exercise 4.1: Working with INNER JOINS

- Please complete this exercise in your Exercise Manual

Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

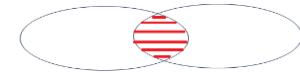
Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

Inner and Outer Join

- The joins we have seen so far have all been *inner* joins
 - The result excludes rows that do *not* satisfy the join condition; i.e., have *no* matching rows in the other table
- Inner joins can be specified with the keyword INNER JOIN
 - This is the default, so the word INNER may be omitted, as we have seen
- It may be desirable to include rows from one table that have no matching row in the other table
 - This is called an *outer join*
 - Columns will contain NULL when there are no matching rows
- Common situations where there are no matching rows:
 - A primary key value with no corresponding foreign key values
 - There should never be foreign key values with no corresponding primary key value, because this would violate the referential integrity rule
 - A foreign key with NULL value



Inner Join Example

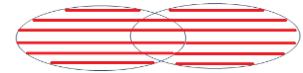
- Inner joins return rows where there is a match between two tables
 - This is the default join type
- Answers business questions like:
 - “I need an employee list with each employee’s department”

■ Try It Now!

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e  
INNER JOIN departments d  
ON e.department_id = d.department_id
```

- How many records were returned?
- Does this seem correct?

Full Outer Join Example



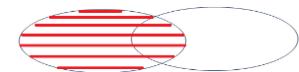
- Full outer joins return all rows from both tables, whether there is a match in the join column or not
 - NULL values are added to columns when there is no match
- Answers business questions like:
 - "I need a list of all the employees in the system and the department they work in. Be sure to include any employees that do not have a department, and any departments without employees"
- **Try It Now! and examine the results**

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d
ON e.department_id = d.department_id
```

Full Outer Join Example (continued)

- How many records were returned for the full outer join?
 - Were any records found with NULL in the first/last name?
 - What do you think these records represent?
 - Were there any records that NULL in the Department fields?
 - If present, what would these records represent?

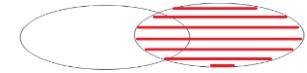
Left Outer Join Example



- Left outer join returns all rows from the table on the left plus any matching rows in the table on the right
 - NULL values are added to columns when there is no match
 - To determine which table is “Left”, look at the `FROM` clause
 - The first table in the `FROM` clause is the left table
- Answers business questions like:
 - “I need the names of all the employees, and the department name (if available) associated with that ID. Include employees that are not assigned to a department.”

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON e.department_id = d.department_id
```

Right Outer Join Example



- Right outer join returns all rows from the table on the right plus any matching rows in the table on the left
 - NULL values are added to columns when there is no match
 - To determine which table is “Right”, look at the `FROM` clause
 - The second table in the `FROM` clause is right table

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
RIGHT OUTER JOIN departments d
ON e.department_id = d.department_id
```

- This example can also be written as a Left Outer Join by changing the left and right tables in the join statement

What Is the Difference Between this LEFT OUTER JOIN and RIGHT OUTER JOIN?

```
SELECT e.first_name, e.last_name, d.department_name  
FROM departments d  
LEFT OUTER JOIN employees e  
ON e.department_id = d.department_id
```

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e  
RIGHT OUTER JOIN departments d  
ON e.department_id = d.department_id
```

Outer Join with Non-Join Condition

- Be careful when combining an outer join and a non-join condition
 - This query only returns results where the department name matches the condition
 - There is no difference between this and the inner join

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON e.department_id = d.department_id
WHERE d.department_name LIKE 'P%';
```

- This query returns all employees, but only matches departments that pass the condition
 - All others are NULL as you would expect in an outer join

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON e.department_id = d.department_id
AND d.department_name LIKE 'P%';
```



HANDS-ON
EXERCISE

45 min

Exercise 4.2: Using OUTER JOINS

- Please complete this exercise in your Exercise Manual

Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

Self JOIN

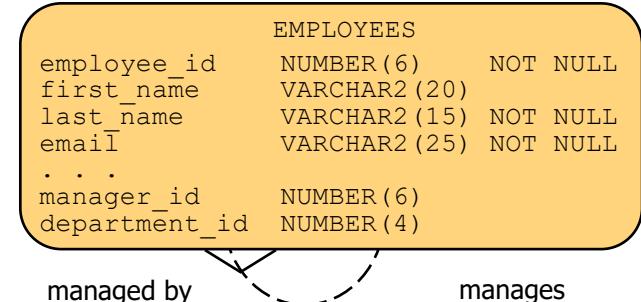
- A table can be joined to itself

- The query “sees” two identical copies of the table
- Table aliases must be used to distinguish between them

- Example of self JOIN:

- For each employee, list the name of the employee and the name of their manager
- manager_id is a foreign key that references the manager's employee_id
- OUTER JOIN is required in order to include employees who have no manager (i.e., NULL in the manager_id column)

```
SELECT e.employee_id, e.last_name, e.manager_id, m.last_name AS manager
FROM employees e
LEFT OUTER JOIN employees m
ON e.manager_id = m.employee_id
ORDER BY e.employee_id;
```



Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

SQL-89 Join Syntax

- The syntax we have used so far was introduced in SQL-92
- Consider this join:

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e  
INNER JOIN departments d  
ON e.department_id = d.department_id
```

- In the older syntax, it would look like this:

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e, departments d  
WHERE e.department_id = d.department_id
```

- There was no standard syntax for outer joins prior to SQL-92
 - The Oracle syntax involved putting (+) next to columns of the optional table in the WHERE clause



SQL-89 Cartesian Join

- What happens if you miss out the join condition?

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e, departments d
```

■ Try it now!

- How many rows are returned?

- Also try these queries

```
SELECT first_name, last_name FROM employees
```

```
SELECT department_name FROM departments
```

- Can you work out what has happened?

Cartesian Join

- In the previous example, a Cartesian Join (or Cartesian Product) was returned
 - We did not define the relationship between the two tables, so all records from the first table were combined with all records from the second table
- A Cartesian Join returns the columns for each table
 - Columns are returned side-by-side
 - If Table 1 has 10 columns, and Table 2 has 5 columns, the result set has 15 columns
 - This query returns a row for each combination between the two tables
 - If one table has 10 rows and the second table has 200 rows, then 2,000 rows would be returned
- There are limited uses for a Cartesian Join
 - Ask your instructor if he/she has ever had to generate one!

Cartesian Join in SQL-92 Syntax

- If you really need a Cartesian Join, you can produce it using the CROSS JOIN

```
SELECT e.first_name, e.last_name, d.department_name  
FROM employees e  
CROSS JOIN departments d
```

- Prefer the SQL-92 syntax in all situations
 - It is more expressive
 - It is harder to inadvertently create Cartesian Joins in complex queries

Chapter Concepts

The Need for Joins

Inner Joins

Outer Joins

Self Joins

Cartesian Joins and SQL-89 Syntax

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The use of `JOINS` to retrieve data from multiple tables
- The difference between `INNER`, `OUTER`, and `CROSS JOIN`
- Joining a table to itself
- Cartesian Join
- SQL-89 vs SQL-92 syntax

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 5: Additional SQL Functions

Chapter Objectives

In this chapter, we will explore:

- Using DATE-related functions
- Miscellaneous functions

Chapter Concepts

DATE Functions

Miscellaneous Functions

Chapter Summary

Date and Time Functions Available

- The date and time functions are:

- ADD_MONTHS
- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE
- EXTRACT(datetime)
- FROM_TZ
- LAST_DAY

- LOCALTIMESTAMP
- MONTHS_BETWEEN
- NEW_TIME
- NEXT_DAY
- NUMTODSINTERVAL
- NUMTOYMINTEGERVAL
- ROUND(date)

- SESSIONTIMEZONE
- SYS_EXTRACT_UTC
- SYSDATE
- SYSTIMESTAMP
- TO_CHAR(datetime)
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ

- TO_DSINTERVAL
- TO_YMINTERVAL
- TRUNC(date)
- TZ_OFFSET

- Most operate on all datetime data types: DATE, TIMESTAMP (and all variations), and INTERVAL types (not covered on this course)

- Exceptions:

- Only DATE: ADD_MONTHS, CURRENT_DATE, LAST_DAY, NEW_TIME, NEXT_DAY
 - If you provide a timestamp, it is converted to a DATE value and a DATE is returned
- MONTHS_BETWEEN returns a number
- ROUND and TRUNC do not accept timestamp or interval values at all

Date and Time Functions: SYSDATE and SYSTIMESTAMP

- SYSDATE is a function call that returns a date datatype
 - The setting on the server where the Oracle database resides
 - To determine the date:

```
SELECT SYSDATE FROM DUAL;
```

```
SYSDATE
```

```
-----  
13-MAY-2019
```

- SYSTIMESTAMP is a function call similar to SYSDATE
 - Returns TIMESTAMP WITH TIMEZONE from the server

```
SELECT SYSTIMESTAMP
```

```
, TO_CHAR(SYSTIMESTAMP, 'YYYY MM DD HH24 MI SS.FF') FROM DUAL;
```

```
SYSTIMESTAMP
```

```
TO_CHAR(SYSTIMESTAMP, 'YYYYMMDD
```

```
-----  
13-MAY-19 17.39.59.086000000 +01:00 2019 05 13 17 39 59.086000
```

Date and Time Functions: ADD_MONTHS

- ADD_MONTHS returns the date input_date plus integer months
- Syntax: ADD_MONTHS(input_date, integer)
 - The return type is always DATE, regardless of the datatype of input_date
- If input_date is the last day of the month or if the resulting month has fewer days than the day component of input_date, then the result is the last day of the resulting month
 - Example:

```
SELECT SYSDATE,  
       ADD_MONTHS(SYSDATE, 2)          AS A,  
       SYSDATE + 18                  AS B,  
       ADD_MONTHS(SYSDATE + 18, 1)    AS C,  
       ADD_MONTHS(SYSDATE + 18, -6)   AS D  
FROM DUAL;
```

SYSDATE	A	B	C	D
13-MAY-2019	13-JUL-2019	31-MAY-2019	30-JUN-2019	30-NOV-2018

Chapter Concepts

DATE Functions

Miscellaneous Functions

Chapter Summary

Miscellaneous Single-Row Functions

- The following single-row functions do not fall into any of the other single-row function categories

Non-XML Functions

BFILENAME
COALESCE
CV
DECODE
DUMP
EMPTY_BLOB
EMPTY_CLOB
EXISTSNODE
GREATEST
LEAST
LNNVL
NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME
NULLIF

NVL
NVL2
ORA_HASH
PRESENTNNV
PRESENTV
PREVIOUS
SYS_CONNECT_BY_PATH
SYS_CONTEXT
SYS_EXTRACT_UTC
SYS_GUID
SYS_TYPEID
UID
USER
USERENV
VSIZE

XML Functions

APPENDCHILDXML
DELETEXML
DEPTH
EXTRACT (XML)
EXISTSNODE
EXTRACTVALUE
INSERTCHILDXML
INSERTXMLBEFORE
PATH
SYS_DBURIGEN
SYS_XMLAGG
SYS_XMLGEN
UPDATEXML
XMLAGG

XMLCDATA
XMLCOLATTVAL
XMLCOMMENT
XMLCONCAT
XMLFOREST
XMLPARSE
XMLPI
XMLQUERY
XMLROOT
XMLSEQUENCE
XMLSERIALIZE
XMLTABLE
XMLTRANSFORM

Dealing with NULLS: COALESCE

- COALESCE returns the first non-null expr in the expression list
 - If all occurrences of expr evaluate to null, then the function returns null
 - COALESCE is ANSI standard
- Syntax: COALESCE(expr1, expr2, ... exprN)

```
SELECT COALESCE(NULL, 2, 3, 4)      AS A,  
       COALESCE(1, NULL, 3, 4)      AS B,  
       COALESCE(NULL, NULL, 3, 4)   AS C  
FROM DUAL;
```

A	B	C
2	1	3

```
SELECT commission_pct, last_name, COALESCE(TO_CHAR(commission_pct), 'No Commission')  
FROM employees  
WHERE employee_id = 100;
```

COMMISSION_PCT	LAST_NAME	COALESCE(TO_CHAR(COMMISSION_PCT), 'NOCOMM')
	King	No Commission

Dealing with NULLS: NVL, NVL2

- NVL, which predates the ANSI COALESCE, is commonly used
- Syntax: NVL(expr1, expr2)
 - Returns expr1 if it is not NULL
 - Returns expr2 otherwise
- Other database vendors have also implemented the NVL function
 - Oracle recommends using COALESCE
- NVL2 is similar
 - NVL2(expr1, expr2, expr3)
 - Returns expr2 if expr1 is NOT NULL, returns expr3 if it is

NVL Example

```
SELECT ename, sal, comm, sal + comm "Total Compensation"  
FROM emp;
```

ENAME	SAL	COMM	Total Compensation
1 SMITH	800	(null)	(null)
2 ALLEN	1600	300	1900
3 WARD	1250	500	1750
4 JONES	2975	(null)	(null)
5 MARTIN	1250	1400	2650

```
SELECT ename, sal, comm, sal + NVL(comm, 0) "Total Compensation"  
FROM emp;
```

ENAME	SAL	COMM	Total Compensation
1 SMITH	800	(null)	800
2 ALLEN	1600	300	1900
3 WARD	1250	500	1750
4 JONES	2975	(null)	2975
5 MARTIN	1250	1400	2650



HANDS-ON
EXERCISE

30 min

Exercise 5.1: Additional SQL Functions

- Please complete this exercise in your Exercise Manual

Chapter Concepts

DATE Functions

Miscellaneous Functions

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Using DATE-related function
- Miscellaneous functions

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 6: Data Manipulation Language

Chapter Objectives

In this chapter, we will explore:

■ Manipulating data using DML:

- INSERT
- UPDATE
- DELETE

■ Transactional Control Statements:

- COMMIT
- ROLLBACK
- SAVEPOINT

Chapter Concepts

INSERT

UPDATE

DELETE

Transactional Control

Chapter Summary

Data Manipulation Language Statements

- Data Manipulation Language (DML) is a classification of SQL
 - Others we are talking about:
 - Data Query Language (DQL)
 - The SELECT statement
 - Data Definition Language (DDL)
 - CREATE, ALTER
 - Transaction Control Language (TCL)
 - COMMIT, ROLLBACK, SAVEPOINT
- DML statements control the values of the data

Adding Data: the INSERT Statement

- DML statements allow you to manipulate the data in any table in your account
 - Or to have permission, if the table is owned by another user
 - If synonyms are not defined, qualify the table name: schema.table
- INSERT, UPDATE, and DELETE statements operate against a set of data
 - A set is zero, 1, or many rows
- The syntax for the INSERT allows either the insertion of one row or an entire set of rows

INSERTing One Row of Data

Syntax:

```
INSERT INTO table_name [ (column_list) ]  
VALUES (value_clause)
```

Rules:

- The values list must map to the column list one for one
- All integrity rules must be met
- If there is a defined default value for a column, it is invoked using the key word:
DEFAULT
 - This table setting is covered later
- The column list is optional
 - Values list must include a value for each column in the table in the order they appear in the table
 - Even if the value is NULL
 - Only use this for ad hoc inserts; always use the column list in production code

INSERTing One Row of Data: Examples

- The following are equivalent:

```
INSERT INTO regions  
VALUES (5, 'Antarctica');
```

```
INSERT INTO regions (region_id, region_name)  
VALUES (5, 'Antarctica');
```

```
INSERT INTO regions (region_name, region_id)  
VALUES ('Antarctica', 5);
```

```
INSERT INTO regions  
VALUES (5/1 ,SUBSTR('Antarctica', 1));
```

INSERTing One Row of Data: More Examples

- Unless a column is mandatory, a value does not have to be inserted
 - Columns for which no value is specified are set to NULL
 - Or to their DEFAULT value

```
INSERT INTO regions (region_id) VALUES (5);
```

```
1 row created.
```

```
SELECT *
FROM regions
WHERE region_id = 5;
```

REGION_ID	REGION_NAME
1	5 (null)

Accepting Default Values for Columns

- If the `region_name` column had a `DEFAULT` value, then it would be used if a value was not inserted
 - Specify the key word `DEFAULT` if the column is in the column list

```
INSERT INTO regions (region_id) VALUES (5);
```

```
INSERT INTO regions (region_id, region_name) VALUES (5, DEFAULT);
```

- The result in both cases would be:

REGION_ID	REGION_NAME
-----	-----
5	Anytown

INSERTing a Set of Rows

Syntax:

```
INSERT INTO table_name [ (column_list) ]  
SELECT_clause
```

Rules:

- The `SELECT` clause can be any legal SQL statement
- The `SELECT` list must map to the column list
- The `SELECT` clause can return 0, 1, or more rows of data
- The other rules are the same as for a single row `INSERT`
- Each `INSERT` statement stands on its own
 - A data integrity violation with 1 row in the set causes the failure of all rows attempting to be inserted

INSERTing a Set of Data: Example

- Add rows from the location table into countries
 - Build the column being used as the primary key using the SUBSTR function

```
INSERT INTO countries (country_id, country_name)
SELECT SUBSTR(location_id, 1, 2), state_province
FROM locations;
```

```
23 rows created.
```

Chapter Concepts

INSERT

UPDATE

DELETE

Transactional Control

Chapter Summary

Maintaining Column Values: The UPDATE Statement

- The `UPDATE` modifies existing rows in the table
 - Changes the values of column(s) in 0, 1, or more rows in a table
- The columns can be set to static expressions
 - Including literals and functions
- Or to the returned values from a `SELECT` statement

UPDATE Syntax: Static Expressions

Syntax:

```
UPDATE table_name  
SET column_name = expression | DEFAULT  
      [ , ... ]  
[ WHERE condition ]
```

Rules:

- The key word *SET* is mandatory and appears only once
- More than one column can be updated
 - The expression must resolve to the same datatype
- The *DEFAULT* for the column can be referenced
- The *WHERE* clause is optional
 - **If not specified, ALL rows of the table will be updated**

UPDATE Statement: Static Examples

```
UPDATE regions  
SET region_name = 'NewTown'  
WHERE region_id = 0;
```

```
SELECT *  
FROM regions  
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
0	NewTown

```
UPDATE regions  
SET region_id = 10 - 9  
, region_name = 'SubTown'  
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
1	SubTown

DEFAULTs and NULLs

The `DEFAULT` keyword can be used if one exists

Be careful with `NULL`

- `NULL` is a value
- `'NULL'` is a literal string

```
UPDATE regions
SET    region_name = NULL
WHERE region_id = 0;
```

```
UPDATE regions
SET    region_name = 'NULL'
WHERE region_id = 0;
```

```
UPDATE regions
SET    region_name = DEFAULT
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
0	(null)

REGION_ID	REGION_NAME
0	

REGION_ID	REGION_NAME
0	NULL

REGION_ID	REGION_NAME
0	Anytown

UPDATEing Using Subqueries

- A column can also be set equal to a subquery

- Syntax:

```
UPDATE table_name  
SET { (column [, column] ...) = (subquery)  
      | [ column = { expr | (subquery) | DEFAULT }  
      ]  
      [, ... ]  
WHERE conditional_clause
```

- Example:

```
UPDATE regions  
SET region_name = (  
    SELECT country_name  
    FROM countries  
    WHERE country_id = 'AR'  
)  
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
0	Argentina

UPDATE Syntax: Rules for Using Subqueries

- The subquery must be placed within parenthesis
- Specify a subquery that returns exactly one row for each row updated
 - If you specify only one column in the *update_set_clause*
 - The subquery can return only one value
- If you specify multiple columns in the *update_set_clause*:
 - The subquery must return as many values as you have specified columns
 - The multiple columns must be placed within parentheses
- If the subquery returns no rows, then the column is assigned a null

```
UPDATE regions
SET (region_id, region_name) = (
    SELECT 10, country_name
    FROM countries
    WHERE country_id = 'AR'
)
WHERE region_id = 0;
```

Chapter Concepts

INSERT

UPDATE

DELETE

Transactional Control

Chapter Summary

Removing Row(s): The DELETE Statement

Syntax:

```
DELETE FROM table  
[WHERE conditional_clause]
```

- The conditional clause specifies which rows are to be removed
- The WHERE clause is optional
 - **If not specified, ALL rows of the table will be deleted**

DELETE Examples

- DELETE is a set statement

```
DELETE FROM emp;
```

15 rows deleted.

```
DELETE FROM emp WHERE sal > 6000;
```

1 row deleted.

```
DELETE FROM emp WHERE sal > 1500;
```

8 rows deleted.

```
DELETE FROM emp WHERE sal > 10000;
```

0 rows deleted.

DML and Integrity Constraints

- DELETE, like any DML statement, must conform to integrity constraints
 - Parent rows cannot be removed unless the CASCADE option is set

```
DELETE FROM regions WHERE region_id = 12;
```

```
1 row deleted.
```

```
DELETE FROM regions;
```

```
ERROR at line 1:
```

```
ORA-02292: integrity constraint (HR.COUNTR_REG_FK)
violated - child record found
```

Chapter Concepts

INSERT

UPDATE

DELETE

Transactional Control

Chapter Summary

Transactional Control

- A transaction is a logical unit of work that comprises one or more SQL statements
 - Must leave the database in a consistent state
- Data Manipulation Language (DML) statements are part of transactional control
 - The ability to make one or more changes to data as a group
 - The first SQL statement automatically starts a transaction
- To make the changes permanent to the database, issue a COMMIT
- To undo the changes, issue a ROLLBACK
 - ROLLBACK will undo all changes to the beginning of the transaction

Undoing Part of a Transaction

- SAVEPOINTs can be issued as part of the transactional stream
 - A savepoint is an intermediate marker that divides the transaction into smaller pieces

■ Syntax:

```
SAVEPOINT my_savepoint;
```

- Allows the session to roll back any DML that was issued after the establishment of the savepoint

ROLLBACK Example

- First, set up the transactional activity in the session

```
INSERT INTO jobs VALUES ('IT_AC_PROG', 'Accounting Programmer', 1000, 1000);  
1 row created.
```

```
SAVEPOINT sp_jobs;  
Savepoint created.
```

```
INSERT INTO jobs VALUES ('IT_PR_PROG', 'Payroll Programmer', 999999, 999999);  
1 row created.
```

```
SELECT job_id, job_title FROM jobs WHERE job_id IN ('IT_AC_PROG', 'IT_PR_PROG');  
JOB_ID      JOB_TITLE  
-----
```

```
IT_AC_PROG Accounting Programmer  
IT_PR_PROG Payroll Programmer
```

ROLLBACK Example (continued)

- Undo the second insert by issuing the ROLLBACK TO... statement

```
ROLLBACK TO sp_jobs;  
Rollback complete.
```

- Test for the existence of the inserts

JOB_ID	JOB_TITLE
IT_AC_PROG	Accounting Programmer

- Note that the Payroll programmer no longer exists

- Finally, issue a session-level rollback and test again

```
ROLLBACK;  
Rollback complete.
```

```
SELECT job_id, job_title FROM jobs WHERE job_id IN ('IT_AC_PROG', 'IT_PR_PROG');  
no rows selected
```

- Where did we roll back to?

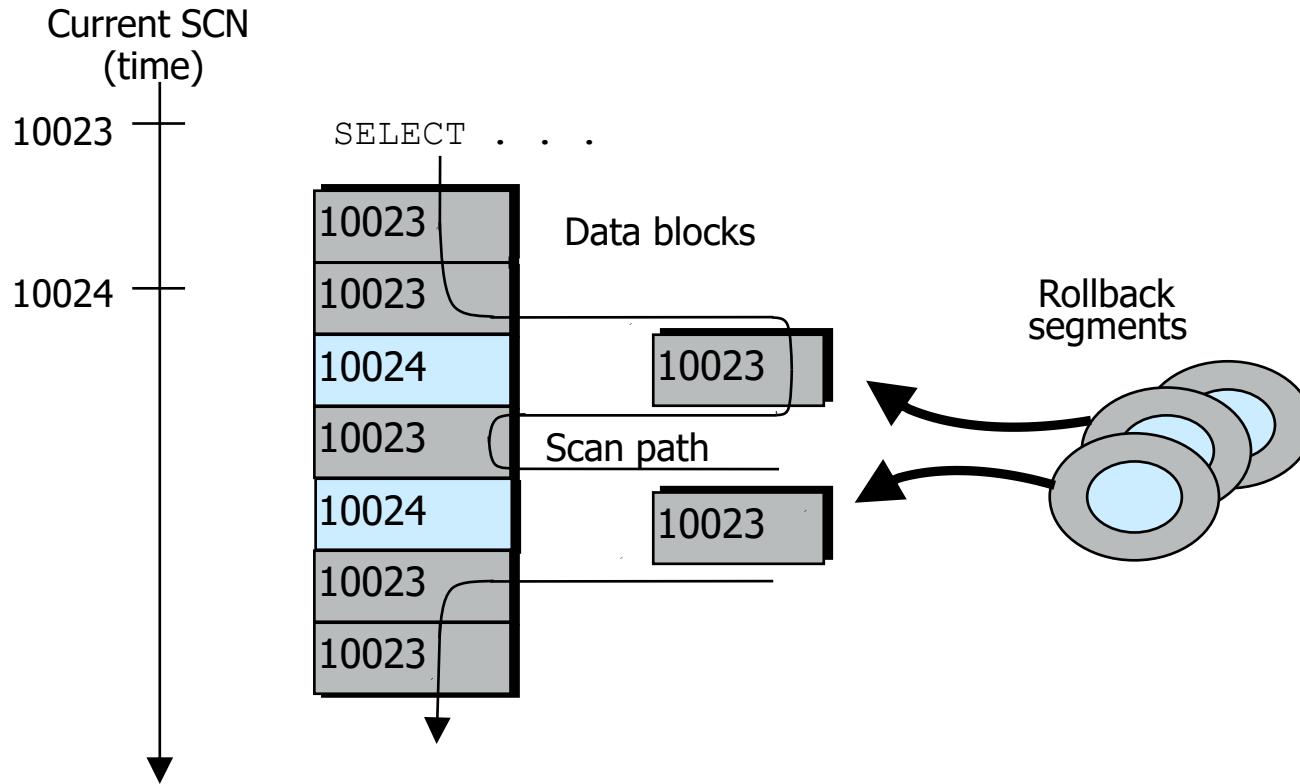
Read Consistency

- Statement-level read consistency
 - A single select is not impacted by database changes during its execution
- Transaction-level read consistency
 - Multiple selects in one transaction are not impacted by database changes
 - Enforced by `SET TRANSACTION READ ONLY`
- Ensures that readers of database data do not have to wait for writers or readers of the same data
- Ensures that writers of database data do not have to wait for readers of the same data
- Automatically enforced by Oracle using rollback segments and the System Change Number (SCN)

Read Consistency Example

- As query enters execution stage, the current SCN is determined
 - SCN is a sequential number assigned to each transaction
 - Recorded in rollback segment and redo log
 - In this example, SCN is 10023
- As query reads, only data blocks with SCN 10023 are used
- Blocks with changed data (more recent SCNs) are reconstructed using the rollback segments
- Reconstructed data is returned to the query
- Query returns all committed data with respect to the SCN recorded at the time execution started
- Committed or uncommitted changes of other transactions that occur during a query's execution are not seen, guaranteeing that a consistent set of data is returned for each query

Read Consistency Example (continued)



SET TRANSACTION READ ONLY Command

- First statement of a read-only transaction
- Remaining statements in the transaction cannot change data in the database
 - Insert, update, and delete are not allowed
- Last statement in the read-only transaction is COMMIT or ROLLBACK

Example of Transaction-Level Read Consistency

- Provide two listings of the `employees` table ordered by:

- Last name
 - Hire date

- Transaction begins

- Establish read consistency for the transaction

```
SET TRANSACTION READ ONLY;
```

- Create the employee listing by last name

```
SELECT *
FROM employees
ORDER BY last_name;
```

- Create the employee listing by hire date

```
SELECT *
FROM employees
ORDER BY hire_date;
```

- Release read consistency

```
ROLLBACK;
```

Advantages and Disadvantages of Read Consistency

Advantage:

- Allows consistency of data throughout a transaction

Disadvantages:

- Performance of SELECT statements is impacted due to the necessity of reconstructing data from ROLLBACK segments
- May get snapshot too old error

Serializable Transactions

- Makes it appear as if there are no other users modifying data in the database
 - Any row read in the transaction is assured to be the same upon a reread
 - Most restrictive form of transaction isolation

- Set at the transaction level with the command

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- Causes all statements in the transaction, including SELECT, to obtain locks

- Improvement over SET TRANSACTION READ ONLY because it allows UPDATE, INSERT, and DELETE

- *Beware:* high cost to performance



Exercise 6.1: Manipulating Data

60 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

INSERT

UPDATE

DELETE

Transactional Control

Chapter Summary

Chapter Summary

In this chapter, we have explored:

■ Manipulating data using DML:

- INSERT
- UPDATE
- DELETE

■ Transactional Control Statements:

- COMMIT
- ROLLBACK
- SAVEPOINT

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 7: Databases with JDBC (Java Database Connectivity)

Chapter Overview

In this chapter, we will explore:

- How Java provides a set of interfaces for connecting to and working with SQL databases
 - Database vendors provide implementations of those interfaces
 - Allows you to write reusable database code
- Why working with databases requires a standard set of steps
 - You will practice these steps and then build them into an object
- Why database security is essential
 - Guard against bad user input
 - SQL injection attacks
- Data Access Object Design Pattern
 - What is the problem we need to solve?
 - What does the implementation look like?
 - What are the trade-offs?

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

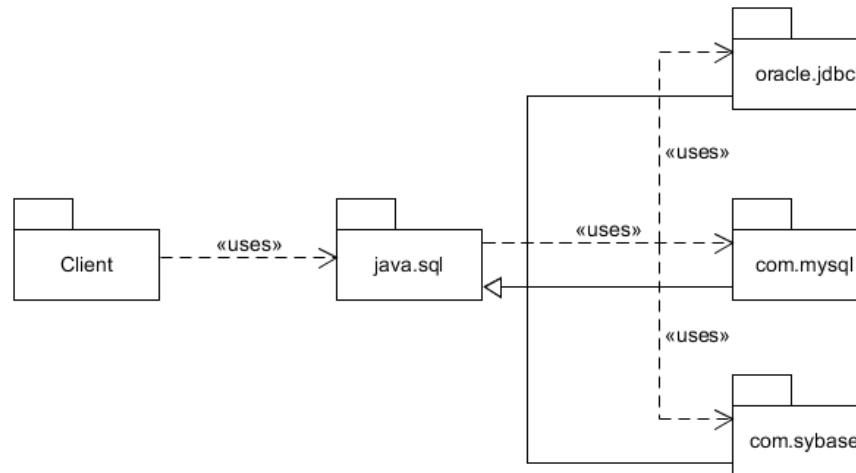
Handling NULL

Enumerated Types

Chapter Summary

java.sql

- Java provides a set of **interfaces** that offer a portable means of accessing databases
 - Java Database Connectivity (JDBC), supplied by `java.sql`
 - Supports standard SQL-92 syntax
 - The same Java code can access Oracle, MySQL, or Sybase database
 - The database vendors provide *drivers* that hook into `java.sql`



Connecting and Executing Queries

- To connect to a database and execute queries, every application has to:
 1. Load the database driver (only required for **very** old versions of Java)
 2. Create a Connection to the database
 3. Create a statement to execute SQL queries
 4. Parse the returned results of database call
 5. Close results and statements
 6. Close Connection
- Errors can occur at **any** of these steps
 - SQLExceptions will be thrown when an error happens

1. Loading and Registering the Driver

■ If using JDBC 4 (Java 6), or beyond, this is not needed:

- Any driver visible to the JVM at start-up is automatically loaded

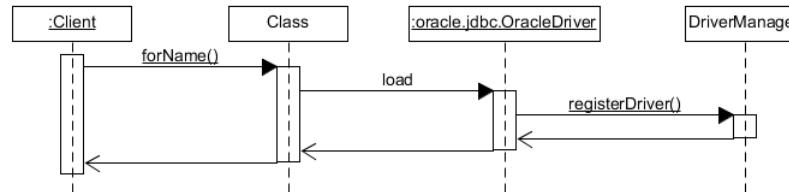
■ The client code has to load up the database vendor's driver

```
Class.forName("oracle.jdbc.OracleDriver");
```

- The driver will then register itself with java.sql.DriverManager

- Only needs to be done once

■ If using MySQL, would register MySQL driver instead:



```
Class.forName("com.mysql.jdbc.Driver");
```

2. Creating Connection to Database

- The DriverManager can create a Connection to the database

```
public Connection getConnection() {  
    String dbUrl = "jdbc:oracle:thin@roifmrwinvm:1521/XE";  
    String user = "scott";  
    String password = "TIGER";  
    Connection conn = null;  
  
    try {  
        conn = DriverManager.getConnection(dbUrl, user, password);  
    } catch (SQLException e) {  
        e.printStackTrace(); // better way coming soon  
    }  
  
    return conn;  
}
```

Use the same settings as you used in SQL Developer

Password is case sensitive

```
// For MySQL  
String dbUrl = "jdbc:mysql://localhost/mydb";  
String user = "root";  
String password = "root";
```

Reading Database Connection Properties

- It is possible to read the database Connection properties from a file

```
Properties properties = new Properties();
properties.load(this.getClass()
    .getClassLoader()
    .getResourceAsStream("db.properties"));

String dbUrl      = properties.getProperty("db.url");
String user       = properties.getProperty("db.username");
String password  = properties.getProperty("db.password");
```

db.url=jdbc:oracle:thin:@localhost:1521/XEPDB1
db.username=scott
db.password=TIGER
db.driver=oracle.jdbc.driver.OracleDriver

db.properties

DataSource

- Opening and closing database Connections is expensive
 - It is more efficient to reuse Connections to the same database
- Many JEE application servers provide an implementation of `javax.sql.DataSource`
 - Most provide a pool of database Connections
 - Requires a Java Naming and Directory (JNDI) service
 - Provided by the application server
 - It is possible to request Connections capable of participating in distributed transactions
- We will use a very simple DataSource for testing our JDBC code
 - This does NOT use a Connection pool

```
DataSource dataSource = new SimpleDataSource();
Connection connection = dataSource.getConnection();
```



Exercise 7.1: Connecting to a Database

20 min

- Complete this exercise described in the Exercise Manual

- Use TDD—verify your code works (i.e., does not throw an exception)

- The dependency for the Oracle database must be in pom.xml

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>18.3.0.0</version>
</dependency>
```

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

JDBC Statements

- The Connection can create a Statement to execute an SQL command
- There are three types of JDBC Statements:

1. Statement
 - Don't use this!
 - Vulnerable to SQL Injection exploits
2. PreparedStatement
 - Always use this for SQL commands
3. CallableStatement
 - Use this for executing a stored procedure in the database



User Inputs into SQL Queries



- Directly embedding user inputs into SQL queries is dangerous

```
public List<Permission> queryPermissionsByUserUnsafe(String user) {  
    String sql = "SELECT perm FROM permissions WHERE username = '" + user + "'";  
    List<Permission> perms = new ArrayList<>();  
    Statement stmt = null;  
    Connection conn = dataSource.getConnection();  
    try {  
        stmt = conn.createStatement();  
        ResultSet rs = stmt.executeQuery(sql);  
        while (rs.next()) {  
            String perm = rs.getString("perm");  
            Permission permission = new Permission(perm);  
            perms.add(permission);  
        }  
    } catch (SQLException e) {  
        // etc  
    }  
}
```

SQL Injection

- Code that directly embeds user inputs lays itself open to SQL injection attacks

```
String sql = "SELECT perm FROM permissions WHERE username = '' + user + ''";
```

- What if the parameter came from user input and someone entered the following String?

```
"Bobby' OR '1'='1"
```

- `sql` would become:

```
SELECT perm FROM permissions WHERE username = 'Bobby' OR '1' = '1'
```

- “Little Bobby Tables”

- <https://xkcd.com/327/>

Preventing SQL Injection

- Preventing SQL injection is simple
 - **Never** create SQL by concatenating string input from the user
 - **Always** use a PreparedStatement to insert the values into the query
- **Important Note:** The parameter indices start with 1 (not 0)!

```
public List<Permission> queryPermissionsByUserSafe(String user) {  
    String sql = "SELECT perm FROM permissions WHERE username = ?";  
    List<Permission> perms = new ArrayList<>();  
    PreparedStatement stmt = null;  
    Connection conn = dataSource.getConnection();  
    try {  
        stmt = conn.prepareStatement(sql);  
        stmt.setString(1, user);  
        ResultSet rs = stmt.executeQuery();  
        // etc
```

3. Perform a Query

SQL calls are executed by a PreparedStatement

- Use the Connection to prepare a PreparedStatement
- Notice that there is no ; at the end of the query string

```
public List<Department> queryDepartmentsByName(String name) {  
    String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";  
    List<Department> depts = new ArrayList<>();  
    PreparedStatement stmt = null;  
    Connection conn = dataSource.getConnection();  
    try {  
        stmt = conn.prepareStatement(sql);  
        stmt.setString(1, dname);  
        ResultSet rs = stmt.executeQuery();  
        // process returned data  
        ...  
    } catch (SQLException e) {  
        e.printStackTrace(); // better way coming soon  
    } finally {  
        ... // IMPORTANT: close connection  
    }  
}
```

4. Parsing Results from SELECT Statements

- `rs.next()` moves to the next row of result set
- `rs.getInt()`,
`rs.getString()`, etc.
retrieve fields from current row
- This example requires a constructor with arguments for `Department`
- It is also a very good idea to define the `hashCode()` and `equals()` methods in the model

```
public List<Department> queryDepartmentsByName(String name) {  
    String sql = "SELECT deptno, dname, loc FROM dept "  
        + "WHERE dname = ?";  
    List<Department> depts = new ArrayList<>();  
    PreparedStatement stmt = null;  
    Connection conn = dataSource.getConnection();  
    try {  
        stmt = conn.prepareStatement(sql);  
        stmt.setString(1, name);  
        ResultSet rs = stmt.executeQuery();  
        while (rs.next()) {  
            int deptNumber = rs.getInt("deptno"); // or rs.getInt(1)  
            String deptName = rs.getString("dname"); // or rs.getString(2)  
            String loc = rs.getString("loc");  
            Department dept = new Department(deptNumber, deptName, loc);  
            depts.add(dept);  
        }  
    } catch (SQLException e) {  
        e.printStackTrace(); // better way coming soon  
    } finally { ... /* close connection */ }  
    return depts;  
}
```

5. Close JDBC Resources

- When done with a ResultSet, Statement, or Connection, close it
 - Closing a Connection automatically closes its Statements and ResultSets
 - But the JDBC driver never automatically closes a Connection
- You need to close every Connection to avoid resource leaks
 - Some databases (e.g., Oracle) keep connections open indefinitely unless explicitly closed
- close() may throw a checked exception—requires additional exception handling

```
try {  
    ... // database operations  
} finally {  
    if (conn != null) {  
        try {  
            conn.close();  
        } catch (SQLException e) {  
            logger.error("can't close connection", e);  
        }  
    }  
}
```

Closing the Connection

- Database Connections are precious resources
 - Only a limited number of Connections can be open at any moment
 - It may not be possible to connect to a database when all Connections are in use
- Opening and closing Connections are expensive operations
 - But keeping Connections open may prevent other users from connecting to the database
- The solution is to use a **DataSource**
 - Most enterprise DataSources define a pool of database Connections
 - Get a Connection by calling `getConnection()` on the DataSource
 - Return a Connection to the pool by calling `close()` on the Connection

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Data Access Object

- Name: Data Access Object (DAO)
 - Not to be confused with DOA
- Problem: There are several to many places in your application that need to communicate with a data source such as a relational database
- Solution: Encapsulate the data source communication code in one object—the Data Access Object. Other parts of the program can call on the DAO to communicate with the data source
- Consequences:
 - The only part of the program that needs to know the data source details is the DAO
 - The rest of the program is insulated from any data source-specific details

DAO Implementation

- The DAO should get a Connection from a DataSource
 - The DAO should only use the Connection to prepare a PreparedStatement
 - The DAO should call close() on the Connection when finished communicating with the database
 - This returns the Connection to the DataSource
 - The DataSource will be responsible for closing the Connection with the database
- DAO methods should map to required database operations
 - How to hide how data objects are being created?
 - How to handle exceptions?

Exception Handling

- Many of the lines of code throw Exceptions
 - Almost all JDBC methods can throw a SQLException
 - For example, `rs.next()`, `rs.close()`, etc.
- Few database-driven applications can work if the database is inaccessible
 - Simply catch the exception and re-throw as a custom `RuntimeException`
 - The Business or Presentation layer should catch and deal with exception gracefully

```
try {  
    // all the database code  
} catch (SQLException e) {  
    logger.error("Cannot execute SQL query for dept: {}", sql, e);  
    throw new DatabaseException("Cannot execute SQL query for dept: " + sql, e);  
}
```

Note that the original exception is “chained” so it appears in the stack trace as “caused by”

Without Try-With-Resources

- Simplifying the exception handling allows us to clean up our code with try-with-resources

```
@Override  
public List<Department> queryDepartmentsByName(String name) {  
    String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";  
    List<Department> depts = new ArrayList<>();  
    PreparedStatement stmt = null;  
    Connection conn = dataSource.getConnection();  
    try {  
        stmt = conn.prepareStatement(sql);  
        ... // execute statement, process result set  
    } catch (SQLException e) {  
        logger.error("Cannot execute SQL query for dept: {}", name, e);  
        throw new DatabaseException("Cannot execute SQL query for dept: " + name, e);  
    } finally {  
        if (conn != null) {  
            try {  
                conn.close();  
            } catch (SQLException e) {  
                logger.error("Cannot execute close connection", e);  
            }  
        }  
        if (stmt != null) { ... /* similar code to close stmt */ }  
    }  
    return depts;  
}
```

Have to declare conn outside the try-catch-finally so we can use it in finally

This finally block is particularly ugly. Exception handling in a finally block is always troublesome because we cannot easily throw an exception without masking any exception from the catch block and we cannot easily tell if the catch block threw an exception!

Using Try-With-Resources

- Any resource that implements AutoCloseable (or Closeable) can be used in a try-with-resources block
 - Resource is automatically closed without needing to write a finally block
 - Allows better scoping of the resource

```
@Override  
public List<Department> queryDepartmentsByNameSimpler(String name) {  
    String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";  
    List<Department> depts = new ArrayList<>();  
    try (Connection conn = dataSource.getConnection();  
        PreparedStatement stmt = conn.prepareStatement(sql)) {  
        stmt.setString(1, name);  
        ... // execute statement, process result set  
    } catch (SQLException e) {  
        logger.error("Cannot execute SQL query for dept: {}", name, e);  
        throw new DatabaseException("Cannot execute SQL query for dept: " + name, e);  
    }  
    return depts;  
}
```

Initialize the resource here. Can have multiple lines with semicolons.

Scope of conn now restricted to the try block

No ugly finally block. Connection and statement are automatically closed.

Logging with SLF4J

- It is often useful to know what is happening in a running production system
 - For troubleshooting; to choose optimizations; to allow detailed application analysis
 - Java logging makes this easy
 - SLF4J: flexible, easy-to-use logging framework
 - <https://www.slf4j.org/apidocs/index.html>
- Log parameters
 - Rather than this: `System.out.println("Compute iteration " + k);`
 - Use this: `logger.debug("Compute iteration {}", k);`
 - Parameters are not evaluated unless the appropriate logging level is enabled
- SLF4J tutorial: <https://www.baeldung.com/slf4j-with-log4j2-logback>
- **Log4j2 vulnerability:** <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-44228>

Using Java Logging

- Log exceptions with `logger.error()`
 - By default, includes the stack trace in the log file

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class EmployeeDao {
    private final Logger logger = LoggerFactory.getLogger(getClass());
    ...
    try {
        ...
    } catch (SQLException ex) {
        logger.error("Cannot execute SQL query for dept {}", name, ex);
        ...
    }
}
```

`logger.error()` logs stack of exception

Dates and Times — Simple Cases (No Time Zone)

- Since Java 8, the preferred representations are from `java.time`
 - `LocalDate` represents a date without time or time zone
 - `LocalTime` represents a time without time zone
 - `LocalDateTime` represents a date and time without time zone
- JDBC provides simple mappings

Java	JDBC (ANSI SQL)	Oracle	Comments
<code>LocalDate</code>	<code>DATE</code>	<code>DATE</code>	Oracle DATE includes time, stripped out by JDBC
<code>LocalTime</code>	<code>TIME</code>	<code>DATE</code> <code>TIMESTAMP</code>	Oracle DATE includes date, stripped out by JDBC
<code>LocalDateTime</code>	<code>TIMESTAMP</code>	<code>DATE</code> <code>TIMESTAMP</code>	<code>TIMESTAMP</code> contains fractional seconds, Oracle DATE does not

Querying Dates and Times

```
String sql = "SELECT * FROM datetimetest";
try (PreparedStatement stmt = conn.prepareStatement(sql)) {
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        LocalDate ld = rs.getDate("date_test").toLocalDate();
        System.out.println(ld);

        LocalTime lt = rs.getTime("time_test").toLocalTime();
        System.out.println(lt);

        LocalDateTime ldt1 = rs.getTimestamp("datetime_test").toLocalDateTime();
        System.out.println(ldt1);

        LocalDateTime ldt2 = rs.getTimestamp("timestamp_test").toLocalDateTime();
        System.out.println(ldt2);
    }
}
```

```
CREATE TABLE datetimetest (
    date_test      DATE,
    time_test      DATE,
    datetime_test  DATE,
    timestamp_test TIMESTAMP
)
```

```
2017-12-31
23:59:59
2017-12-31T23:59:59
2017-12-31T23:59:59.123456
```

Inserting Dates and Times

```
String sql = "INSERT INTO datetimetest VALUES (?, ?, ?, ?, ?);  
try (PreparedStatement stmt = conn.prepareStatement(sql)) {  
    LocalDate ld = LocalDate.of(2017, 12, 31);  
    stmt.setDate(1, Date.valueOf(ld));  
  
    LocalTime lt = LocalTime.of(23, 59, 59);  
    stmt.setTime(2, Time.valueOf(lt));  
  
    LocalDateTime ldt1 = LocalDateTime.of(ld, lt);  
    stmt.setTimestamp(3, Timestamp.valueOf(ldt1));  
  
    LocalDateTime ldt2 = LocalDateTime.of(2017, 12, 31, 23, 59, 59, 123456000);  
    stmt.setTimestamp(4, Timestamp.valueOf(ldt2));  
  
    stmt.executeUpdate();  
}
```

```
CREATE TABLE datetimetest (  
    date_test          DATE,  
    time_test          DATE,  
    datetime_test      DATE,  
    timestamp_test     TIMESTAMP  
)
```

The JDBC types are `java.sql.Date`,
`java.sql.Time` and `java.sql.Timestamp`

Working with BigDecimal in JDBC

■ Typical insert or update

```
stmt.setLong(1, employee.getId());
stmt.setString(2, employee.getName());
stmt.setBigDecimal(3, employee.getSalary());
stmt.executeUpdate();
```

■ Typical select

```
stmt.setLong(1, department);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    Employee employee = new Employee(rs.getLong("EMPNO"),
                                      rs.getString("ENAME"),
                                      rs.getBigDecimal("SAL"));
    employees.add(employee);
}
```

What to Test — SELECT

- Need to verify that a database operation succeeded
 - Usually, a DAO method converts a result set into one or more objects
 - Test goals: Did the query create valid objects? Were the right objects created?

```
SELECT ... FROM dept ... ORDER BY deptno
```

Add ORDER BY so results are in a predictable order

```
private Department dept10 = new Department(10, "ACCOUNTING", "NEW YORK");
private Department dept40 = new Department(40, "OPERATIONS", "BOSTON");

@Test
public void testQueryAllDepartments() {
    List<Department> depts = dao.queryAllDepartments();
    assertEquals(4, depts.size());
    assertEquals(dept10, depts.get(0)); // verify first item
    assertEquals(dept40, depts.get(depts.size() - 1)); // verify last item
}
```

```
public class Department {
    public boolean equals(Object o) { ... }
    public String toString() { ... }
    ...
}
```

assertEquals() uses
equals() for comparisons and
toString() for error messages

Exercise 7.2: Creating Objects from Database Query



60 min

- Complete this exercise described in the Exercise Manual

- Use JDBC—make sure you complete all the steps to get data from the database
- Use TDD—how will you test your objects are created correctly?
- Use Eclipse for debugging
 - Starting at the top of a stack trace, find your own code within the stack trace and click it to take you to a location very close to the error
 - Read the entire error message to understand what went wrong

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Reading Database NULL in JDBC

- JDBC's handling of NULL columns is not always intuitive
- Example: In PRODUCT table, SHIPPING_WEIGHT is a nullable NUMERIC column
- Product class defines a nullable shippingWeight property:

```
public class Product {  
    private Double shippingWeight;  
    public void setShippingWeight(Double weight) { shippingWeight = weight; }
```

Properties of type Double can be null

- Task: read SHIPPING_WEIGHT from DB and set a Product's shippingWeight property
 - But if column value is NULL, set shippingWeight to null
 - First attempt:

```
Double weight = rs.getDouble("shipping_weight");  
product.setShippingWeight(weight);
```

Doesn't work! If column is NULL,
getDouble() returns 0.0

Reading Database NULL in JDBC (continued)

- For Java primitives, NULL maps to 0 for numeric types, false for boolean
 - To find out if a column really is NULL, use `rs.wasNull()` after getting the column
 - To allow null values, replace primitive fields with wrapper classes (`Double`, `Boolean`)

```
Double weight = rs.getDouble("shipping_weight");
if (rs.wasNull()) {
    weight = null;
}
product.setShippingWeight(weight);
```

- For objects (`String`, `Date`, `BigDecimal`, etc.), JDBC maps database NULL to Java null
 - Test value before calling conversion methods to avoid `NullPointerException`

```
LocalDate hireDate = null;
Date dbHireDate = rs.getDate("hiredate");
if (dbHireDate != null) {
    hireDate = dbHireDate.toLocalDate();
}
employee.setHireDate(hireDate);
```

Don't call `toLocalDate()` if
hiredate column was NULL

Writing Database NULL in JDBC

- Statement set methods that accept object types interpret Java null as database NULL

```
stmt.setDate(5, null);
```

Okay

- Statement has special methods that accept Java primitives and set a column to NULL

```
if (employee.getComm() == null) {  
    stmt.setNull(7, java.sql.Types.NUMERIC);  
} else {  
    stmt.setDouble(7, employee.getComm());  
}
```

This is the JDBC type
(based on ANSI SQL) that
maps to Oracle NUMBER

- If a getter method might return null, test the value before setting a column

```
stmt.setDouble(7, employee.getComm());
```

Generates an NPE if
getComm() returns null

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Handling enum

- To store an enum property in a database table, you could store the enum's string value
 - For String/VARCHAR2 columns, just use the enum name
 - Use standard methods `toString()` and `valueOf()`

```
public enum PerformanceReviewResult {  
    BELOW,  
    AVERAGE,  
    ABOVE  
}  
  
public class Employee {  
    private PerformanceReviewResult review;  
    public PerformanceReviewResult getPerformanceReviewResult() {  
        return review;  
    }  
}
```

```
String perfRev = rs.getString("perf_rev_name"); // "BELOW", "AVERAGE", "ABOVE"  
PerformanceReviewResult revResult = PerformanceReviewResult.valueOf(perfRev);
```

```
PerformanceReviewResult review = employee.getPerformanceReviewResult();  
stmt.setString(3, review.toString()); // "BELOW", "AVERAGE", "ABOVE"
```

Handling enum (continued)

- More commonly, enums are stored in the database as numeric values
 - enum needs constructor, static factory method, and getter method

```
public enum PerformanceReviewResult {  
    BELOW(1), AVERAGE(3), ABOVE(5);  
    private int code;  
    private PerformanceReviewResult(int code) {  
        this.code = code;  
    }  
    public static PerformanceReviewResult of(int code) {  
        for (PerformanceReviewResult revRes :  
            PerformanceReviewResult.values()) {  
            if (revRes.getCode() == code) {  
                return revRes;  
            }  
        }  
        throw new IllegalArgumentException("bad code: " + code);  
    }  
    public int getCode() { return code; }  
}
```

Call enum constructor with integer argument

Private constructor

Static factory method converts integer to enum value

```
int perfRev = rs.getInt("perf_rev_code");  
PerformanceReviewResult revResult =  
    PerformanceReviewResult.of(perfRev);
```

```
PerformanceReviewResult review =  
    employee.getPerformanceReviewResult();  
stmt.setInt(3, review.getCode()); // 1, 3, 5
```

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How Java provides a set of interfaces for connecting to and working with SQL databases
 - Database vendors provide implementations of those interfaces
 - Allows you to write reusable database code
- Why working with databases requires a standard set of steps
 - You will practice these steps and then build them into an object
- Why database security is essential
 - Guard against bad user input
 - SQL injection attacks
- Data Access Object Design Pattern
 - What is the problem we need to solve?
 - What does the implementation look like?
 - What are the trade-offs?

Key Points

- Java interfaces provide standard, portable way to access SQL databases
- Accessing databases provides an effective way to create business objects
 - Code for working with databases should be encapsulated
- Never concatenate user inputs directly to SQL queries
 - You cannot trust data directly given by users
 - Use `PreparedStatement` to avoid SQL injection attacks
- Create a data access object (DAO) that rest of code base will use
 - DAO consolidates all database access in a single class
 - Receives a `Connection` from a `DataSource` in each method
 - Methods of DAO map to database operations

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 8: Updating Databases

Chapter Overview

In this chapter, we will explore:

- Manipulating data using Data Manipulation Language (DML):
 - INSERT
 - UPDATE
 - DELETE
- Executing DML commands with JDBC
- Managing transactions
 - ACID
 - Isolation levels
- Using transactions in testing
 - Use a TransactionManager
 - Start a transaction before each test
 - Rollback the transaction after each test

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

Data Manipulation Language Statements

SQL Review

■ Data Manipulation Language (DML) is a classification of SQL

- DML statements control the values of the data
- Manipulating data using DML:
 - INSERT
 - UPDATE
 - DELETE

■ Others we are talking about:

- Data Query Language (DQL)
 - The SELECT statement
- Data Definition Language (DDL)
 - CREATE, ALTER
- Transaction Control Language (TCL)
 - COMMIT, ROLLBACK, SAVEPOINT

Adding Data: the INSERT Statement

SQL Review

- DML statements allow you to manipulate the data in any table in your account
 - If synonyms are not defined, qualify the table name: scott.emp
- INSERT, UPDATE, and DELETE statements operate against a set of data
 - A set is zero, 1, or many rows
- The syntax for the INSERT allows either the insertion of one row or an entire set of rows

INSERTing One Row of Data

SQL Review

Syntax:

```
INSERT INTO table_name [ (column_list) ]  
VALUES (value_clause)
```

Rules:

- The values list must map to the column list one for one
- All integrity rules must be met
- If there is a defined default value for a column, it is invoked using the key word:
DEFAULT
- The values list must include a value for each column in the table in the order they appear in the column list
 - Even if the value is NULL

INSERTing One Row of Data: Examples

SQL Review

- The following are equivalent:

```
INSERT INTO regions  
VALUES (5, 'Antarctica');
```

```
INSERT INTO regions (region_id, region_name)  
VALUES (5, 'Antarctica');
```

```
INSERT INTO regions (region_name, region_id)  
VALUES ('Antarctica', 5);
```

```
INSERT INTO regions  
VALUES (5/1 ,SUBSTR('Antarctica', 1));
```

Executing SQL Statements with JDBC

- When executing a statement, consider if the SQL statement returns results:
 - If yes (example: SELECT statements):
 - Use `stmt.executeQuery()`
 - Process the `ResultSet` that is returned
 - If no (example: INSERT, UPDATE, DELETE statements):
 - Use `stmt.executeUpdate()`
 - No `ResultSet` is returned
 - The number of rows affected by the command is returned

Insert with JDBC Example

```
public void addNewDepartment(Department dept) {  
    String sql = "INSERT INTO dept(deptno, dname, loc) VALUES (?, ?, ?);  
  
    try (Connection conn = getConnection();  
        PreparedStatement stmt = conn.prepareStatement(sql)) {  
        stmt.setInt (1, dept.getDeptNumber());  
        stmt.setString(2, dept.getDeptName());  
        stmt.setString(3, dept.getLocation());  
  
        stmt.executeUpdate();  
    }  
    catch (SQLException e) {  
        logger.error("Cannot insert into dept {}", sql, e);  
        throw new DatabaseException("Cannot insert into dept " + sql, e);  
    }  
}
```

Inserting Values into DATE Columns

- SQL DATE type maps to `java.sql.Date`
- Java date types must be converted to `java.sql.Date` before inserting into a table
 - `java.util.Date`
 - `java.time.LocalDate`

```
import java.sql.Date;  
...  
public void addNewEmployee(Employee emp) {  
    PreparedStatement stmt = ...  
    ...  
    Date hireDateSql = Date.valueOf(emp.getHireDate());  
    stmt.setDate(4, hireDateSql);  
}
```

The diagram illustrates the conversion of a `LocalDate` object to a `java.sql.Date` object. It shows the code snippet within a green-bordered box. A callout arrow originates from the text "java.sql.Date" and points to the `import java.sql.Date;` statement. Another callout arrow originates from the text "LocalDate" and points to the `Date.valueOf(emp.getHireDate());` method call.

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

Transaction Review

- Transactions are a series of operations that should either all succeed or be all rolled back
 - For example: an account transfer; if deposit to an account fails, money should not be debited from originating account
- Transactions must pass the **ACID** test:
 - **A**tomic
 - All succeed or all fail
 - **C**onsistent
 - After transaction, data should not be corrupted
 - **I**solated
 - The same data should not be concurrently updated by separate transactions
 - Several isolation levels are available (discussed later)
 - **D**urable
 - After a transaction is complete, it cannot be lost in the event of a system crash

Isolation Level

- Can set the isolation level before starting a transaction
 - If underlying driver and database support that isolation level

```
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

- The isolation levels supported are shown below
 - The safer levels are also slower

TRANSACTION_SERIALIZABLE	Prevents phantom reads, non-repeatable reads, and dirty reads. But subject to database deadlocks.
TRANSACTION_REPEATABLE_READ	Phantom reads are possible (may see rows inserted after transaction was started)
TRANSACTION_READ_COMMITTED	Phantom reads and non-repeatable reads possible (may see data changed after transaction was started)
TRANSACTION_READ_UNCOMMITTED	It is possible to read data that has been entered by another transaction, but not yet committed. Known as a "dirty read".

AutoCommit

- In JDBC, the Connection object manages a transaction
- By default, the JDBC Connection is in AutoCommit mode
 - Every statement is committed immediately
 - What if you need to combine statements so that they execute as a block?
 - Either all the statements succeed or none of them do
 - All the statements need to be executed in a transaction

Running Statements in a Transaction

- To run multiple statements within a transaction:

1. Turn off Connection's AutoCommit
2. Execute the various statements
3. Commit if all statements succeed
4. React to a failure by rolling back

```
Connection conn = ...;
conn.setAutoCommit(false); // 1. turn off auto-commit
try {
    ... // 2. perform database operations
    conn.commit(); // 3. commit if all operations succeed
}
catch (SQLException ex) {
    conn.rollback(); // 4: rollback if an operation fails
    throw new DatabaseException(...);
}
```

- What object should manage the transaction?

DAO Design Principles

- The Data Access Object (DAO) should:
 - Encapsulate the database operations
 - Use a DataSource to obtain a database Connection
 - Use a PreparedStatement in its database operations
 - **Participate** in database transactions
 - Perhaps with other DAO operations
 - Call the Connection close() method when its method has its database operations

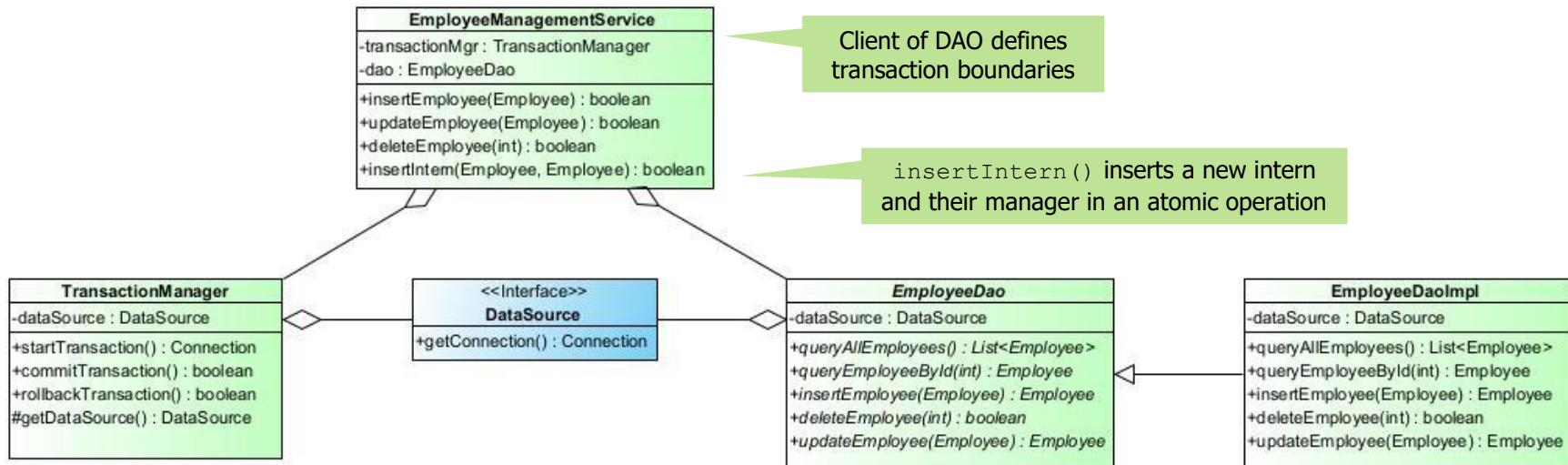
- The Data Access Object should **NOT**:
 - Start, commit, or roll back a transaction
 - There should be no transaction management in the DAO

Managing a JDBC Transaction

- If the DAO does not manage the transaction, who does manage it?
- In an enterprise application, we will need a TransactionManager
 - Start the transaction
 - Commit the transaction
 - Roll back the transaction, in case of an error

Transaction Manager

- TransactionManager lives up to its name:
 - By starting, committing, or rolling back a transaction
 - Before calling a DAO method, client of DAO uses TransactionManager to start a transaction
 - DAO will not be aware of the TransactionManager



TransactionManager Definition

```
public class TransactionManager {  
    private DataSource dataSource;  
  
    public TransactionManager(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    public void startTransaction() {  
        try {  
            Connection connection = dataSource.getConnection();  
            connection.setAutoCommit(false);  
        } catch (SQLException e) {  
            throw new DatabaseException("Unable to begin a transaction", e);  
        }  
  
        public void rollbackTransaction() { ... connection.rollback(); ...}  
  
        public void commitTransaction() { ... connection.commit(); ...}  
    }  
}
```

Service Class with Transaction Management

- This service class uses `TransactionManager` to define atomic database operations

Service without transaction management

```
class EmployeeManagementService {  
    private EmployeeDao dao;  
  
    ...  
  
    public void insertIntern(Employee intern,  
                            Employee sponsor) {  
        dao.insertEmployee(sponsor);  
        dao.insertEmployee(intern);  
    }  
    ...
```

If second insert fails, first
insert will still be committed

Service with transaction management

```
class EmployeeManagementService {  
    private EmployeeDao dao;  
    private TransactionManager transactionMgr;  
  
    ...  
  
    public void insertIntern(Employee intern,  
                            Employee sponsor) {  
        try {  
            transactionMgr.startTransaction();  
            dao.insertEmployee(sponsor);  
            dao.insertEmployee(intern);  
            transactionMgr.commitTransaction();  
        } catch (Exception e) {  
            transactionMgr.rollbackTransaction();  
            throw new DatabaseException(e);  
        }  
    }  
}
```

If either insert fails,
transaction will be rolled back

Using TransactionManager to Test the DAO

- The JUnit tests will use the TransactionManager to manage the transaction
- We will use two Spring libraries to help write tests for the DML operations
 - JdbcTestUtils
 - Requires a database Connection
 - JdbcTemplate
 - Performs database queries
- Our projects will use the DbTestUtils class
 - Defines some helper methods
 - Uses the Spring JdbcTestUtils library

```
public class EmployeeDaoTest {  
    private TransactionManager txManager;  
    private SimpleDataSource dataSource;  
    private Connection connection;  
    private EmployeeDao dao;  
    private DbTestUtils dbTestUtils;  
    private JdbcTemplate jdbcTemplate;  
  
    @BeforeEach  
    void setUp() throws Exception {  
        dataSource = new SimpleDataSource();  
        dao = new EmployeeDaoOracleImpl(dataSource);  
        txManager = new TransactionManager(dataSource);  
        txManager.startTransaction();  
  
        connection = dataSource.getConnection();  
        dbTestUtils = new DbTestUtils(connection);  
        jdbcTemplate = dbTestUtils.initJdbcTemplate();  
    }  
  
    @AfterEach  
    void tearDown() throws Exception {  
        txManager.rollbackTransaction();  
        dataSource.shutdown();  
    }  
}
```

Start transaction before each test

Roll back transaction after each test

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

How Do You Test JDBC Code?

- The purpose of a Unit Test is to test just one unit of the application
 - The System Under Test (SUT) is a single class
 - Isolate SUT from other parts of the system
 - Especially from external components
- When unit testing integration layer:
 - Implementation depends on the database
 - Need to test together with SQL queries
- When unit testing business layer:
 - Isolate from database
 - Use hard-coded data rather than load from database
- When testing system integration:
 - Test progressively larger parts of the system
 - Verify different components work together correctly

Testing a Database INSERT

- We need to know that a database operation succeeded
 - When testing a DAO method that performs an `INSERT`
 - Usually the DAO inserts an object's data into a new database row (or rows)
- Typically, the test will perform the following steps:
 1. Start a transaction
 2. Check the new data is not present before the `INSERT`
 3. Perform the `INSERT`
 4. Check the new data is present after the `INSERT` completes
 5. Roll back the transaction
- Best practice: don't verify a DAO method using other DAO methods
 - Example: don't call `dao.getEmployee()` to verify `dao.insertEmployee()`
 - Instead, verify with Spring library test methods:
 - `JdbcTestUtils.countRowsInTable(jdbcTemplate, table)`
 - `JdbcTestUtils.countRowsInTableWhere(jdbcTemplate, table, where-condition)`

Testing with JdbcTestUtils

- The transaction is started in @BeforeEach
- The transaction is rolled back in @AfterEach

```
@Test  
void testInsertEmployee() throws SQLException {  
    int oldSize = Jdbc TestUtils.countRowsInTable(jdbcTemplate, "emp");  
  
    assertEquals(0, Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "emp", "empno = 8000"));  
  
    Employee new8000 = new Employee(8000, "HEYES", "ANALYST", 7934, LocalDate.of(1980, 12, 17),  
        new BigDecimal("500.00"), null, 10);  
  
    dao.insertEmployee(new8000);  
  
    assertEquals(oldSize + 1, Jdbc TestUtils.countRowsInTable(jdbcTemplate, "emp"));  
  
    assertEquals(1, Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "emp", "empno = 8000"));  
}
```

Get EMP table size before insert

Verify there's no employee with id 8000

Generated query: SELECT COUNT(*) FROM emp WHERE empno = 8000

Call the method under test

Verify EMP table size increased by 1

Verify employee 8000 is now present

Testing with JdbcTestUtils (continued)

- A more thorough test verifies all inserted values, not just the id

```
@Test  
void testInsertEmployee() throws SQLException {  
    ...  
    Employee new8000 = new Employee(8000, "HEYES", "ANALYST", 7934, LocalDate.of(2023, 2, 1),  
        new BigDecimal("90000.00"), null, 10);  
    dao.insertEmployee(new8000);  
  
    assertEquals(1, JdbcTestUtils.countRowsInTableWhere(jdbcTemplate, "emp", """  
        empno = 8000  
        AND ename = 'HEYES'  
        AND job = 'ANALYST'  
        AND mgr = 7934  
        AND hiredate = '1-FEB-23'  
        AND sal = 90000.00  
        AND comm IS NULL  
        AND deptno = 10  
    """));  
}
```

Start text block

Use default Oracle date format
to avoid explicit conversion



HANDS-ON
EXERCISE

60 min

Exercise 8.1: Inserting a Record with JDBC

- Complete this exercise described in the Exercise Manual

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

Removing Row(s): The DELETE Statement

SQL Review

Syntax:

```
DELETE FROM table  
[WHERE conditional_clause]
```

- The conditional clause specifies which rows are to be removed
- The WHERE clause is optional
 - **If not specified, ALL rows of the table will be deleted**

DELETE Examples

SQL Review

- DELETE is a set statement

```
DELETE FROM emp;
```

15 rows deleted.

```
DELETE FROM emp WHERE sal > 6000;
```

1 row deleted.

```
DELETE FROM emp WHERE sal > 1500;
```

8 rows deleted.

```
DELETE FROM emp WHERE sal > 10000;
```

0 rows deleted.

DML and Integrity Constraints

SQL Review

- `DELETE`, like any DML statement, must conform to integrity constraints
 - Parent rows cannot be removed unless the `CASCADE` option is set

```
DELETE FROM regions WHERE region_id = 12;
```

1 row deleted.

```
DELETE FROM regions;
```

ERROR at line 1:

ORA-02292: integrity constraint (HR.COUNTR_REG_FK)
violated - child record found

DELETE with JDBC Example

```
public void deleteDepartment(int deptNumber) {  
    String sql = "DELETE FROM dept WHERE deptno = ?";  
  
    try (Connection conn = getConnection();  
        PreparedStatement stmt = conn.prepareStatement(sql)) {  
        stmt.setInt(1, deptNumber);  
        stmt.executeUpdate();  
    } catch (SQLException e) {  
        logger.error("Cannot delete from dept {}", sql, e);  
        throw new DatabaseException("Cannot delete from dept " + sql, e);  
    }  
}
```

Testing DELETE

```
@Test
void testDeleteDepartment() throws SQLException {
    int oldSize = countRowsInTable(jdbcTemplate, "dept");
    int id = 10;                                Get current size of DEPT table
                                                // delete a Department
    dao.deleteDepartment(id);

    int newSize = countRowsInTable(jdbcTemplate, "dept");
    assertEquals(oldSize - 1, newSize);           Verify DEPT table has one less row
                                                int rowsWithDeletedId = countRowsInTableWhere(jdbcTemplate, "dept", "deptno = " + id);
    assertEquals(0, rowsWithDeletedId);           Verify correct row was deleted
}
```

Maintaining Column Values: The UPDATE Statement

SQL Review

- The `UPDATE` command modifies existing rows in the table
 - Changes the values of column(s) in 0, 1, or more rows in a table
- The columns can be set to static expressions
 - Including literals and functions
- Or to the returned values from a `SELECT` statement

UPDATE Syntax: Static Expressions

SQL Review

Syntax:

```
UPDATE table_name  
SET column_name = expression | DEFAULT  
      [ , ... ]  
[ WHERE condition ]
```

Rules:

- The key word *SET* is mandatory and appears only once
- More than one column can be updated
 - The expression must resolve to the same datatype
- The *DEFAULT* for the column can be referenced
- The *WHERE* clause is optional
 - **If not specified, ALL rows of the table will be updated**

UPDATE Statement: Static Examples

SQL Review

```
UPDATE regions  
SET region_name = 'NewTown'  
WHERE region_id = 0;
```

```
SELECT *  
FROM regions  
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
-----------	-------------

-----	-----
-------	-------

0	NewTown
---	---------

```
UPDATE regions  
SET region_id = 10 - 9  
, region_name = 'SubTown'  
WHERE region_id = 0;
```

REGION_ID	REGION_NAME
-----------	-------------

-----	-----
-------	-------

1	SubTown
---	---------

UPDATE with JDBC Example

```
public Department updateDepartment(Department dept) {  
    String sql = "UPDATE dept SET dname = ?, loc = ? WHERE deptno = ?";  
    try (Connection connection = dataSource.getConnection();  
        PreparedStatement stmt = connection.prepareStatement(sql)) {  
        stmt.setString(1, dept.getName());  
        stmt.setString(2, dept.getLocation());  
        stmt.setInt(3, dept.getNumber());  
  
        stmt.executeUpdate();  
    } catch (SQLException ex) {  
        throw new DatabaseException("Unable to update Department with id=" + dept.getNumber(), ex);  
    }  
  
    return dept;  
}
```

Testing UPDATE

```
@Test
void testUpdateDepartment() {
    int expectedRows = Jdbc TestUtils.countRowsInTable(jdbcTemplate, "dept");
    String whereCondition = "deptno = 10 and dname = 'Updated' and loc = 'Neverland'";
    int rowCount = Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "dept", whereCondition);
    assertEquals(0, rowCount);

    Department dept10 = new Department(10, "Updated", "Neverland");

    // call the method under test
    dao.updateDepartment(dept10);

    // verify the update did not change the row count
    int actualRows = Jdbc TestUtils.countRowsInTable(jdbcTemplate, "dept");
    assertEquals(expectedRows, actualRows);

    // verify department 10's name and location were updated
    rowCount = Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "dept", whereCondition);
    assertEquals(1, rowCount);
}
```

Testing for Exceptions

- The DAO methods use a try-catch structure
 - Throws a DatabaseException if an error condition occurs
- We should test that the DAO actually throws an exception
 - When an error occurs
 - Also referred to as a “negative test”
- To do this, we need to cause an error to occur in our test
- One way to do this is to cause a data integrity violation to occur
 - Attempting to delete a parent row when a child row has a foreign key to the parent
 - Attempting to insert a new row with the same primary key value as an existing row
 - Attempting to insert or update a record with data that violates a database constraint

Testing Exceptions

```
@Test
void testInsertDuplicatePrimaryKeyThrowsException() {
    int duplicatePrimaryKey = 7369;
    LocalDate hireDate = LocalDate.parse("1980-12-17");

    assertThrows(DatabaseException.class, () -> {
        Employee upd7369 = new Employee(duplicatePrimaryKey, "HEYES", "ANALYST", 7934, hireDate,
                                         new BigDecimal("500.00"), new BigDecimal("2"), 10);
        dao.insertEmployee(upd7369);
    });
}
```



HANDS-ON
EXERCISE

Exercise 8.2: Updating and Deleting Records

60 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

Join Example

SQL Review

- Consider the locations and departments tables in the hr database

LOCATIONS

LOCATION_ID	CITY
1700	Seattle
2500	Oxford
2700	Munich

DEPARTMENTS

LOCATION_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1700	30	Purchasing
1700	90	Executive
1700	110	Accounting
1700	120	Treasury
2500	80	Sales
2700	70	Public Relations

- Each department belongs to a particular location
 - We would like to list the departments and include the names of their locations

LOCATION_ID	CITY	DEPARTMENT_ID	DEPARTMENT_NAME
2700	Munich	70	Public Relations
1700	Seattle	90	Executive
1700	Seattle	110	Accounting
1700	Seattle	120	Treasury
2500	Oxford	80	Sales
1700	Seattle	30	Purchasing
...			

Join Example (continued)

SQL Review

- The following query will produce the result on the previous slide:

```
SELECT locations.location_id  
      , locations.city  
      , departments.department_id  
      , departments.department_name  
  FROM departments  
 JOIN locations  
    ON departments.location_id = locations.location_id;
```

Explanation:

- A join returns columns from more than one table
 - In this case, we need columns from two tables
- The `ON` condition specifies how the rows in the tables relate to one another
- The column names have prefixes to specify the table in which each column is located
 - Prefixes will be discussed in more detail later

Mapping One-to-One Relationships to Java

- Assume departments and locations tables have a one-to-one relationship
- Corresponding Java classes contain references to each other
 - At least one class needs a setter method for the other class

```
public class Department {  
    private Location loc; Reference to Location  
    public Department(..., Location loc) {  
        ...  
        this.loc = loc;  
    }  
}
```

```
public List<Department> getDepartments() {  
    ...  
    ResultSet rs = stmt.executeQuery();  
    while (rs.next()) {  
        long locId = rs.getInt("location_id");  
        long deptId = rs.getInt("deptno");  
        ...  
        Location location = new Location(locId, ...);  
    }  
}
```

```
public class Location {  
    private Department dept; Reference to Department  
    public Location(int id, ...) {  
        this.id = id;  
    }  
    public void setDepartment(Department d) {  
        this.dept = d;  
    }  
}
```

```
Department department =  
    new Department(deptId, ..., location);  
...  
location.setDepartment(department);  
...  
Add location to department  
Add department to location
```

Another Join Example

SQL Review

- For each employee, list the employee id, first name, last name, job id, and job title
 - job_title is a column in the jobs table
 - The other columns are located in the employees table

```
SELECT employees.employee_id  
      , employees.first_name  
      , employees.last_name  
      , employees.job_id  
      , jobs.job_title  
  FROM employees  
JOIN   jobs  
ON     employees.job_id = jobs.job_id;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
100	Steven	King	AD_PRES	President
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
109	Daniel	Faviet	FI_ACCOUNT	Accountant
108	Nancy	Greenberg	FI_MGR	Finance Manager
...				

JOIN and WHERE

SQL Review

- It is possible to combine a WHERE condition with a JOIN
 - Example: Restrict the previous example to include only the job id of FI_ACCOUNT

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
  FROM employees e
 JOIN jobs j
    ON e.job_id = j.job_id
 WHERE e.job_id = 'FI_ACCOUNT';
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
109	Daniel	Faviet	FI_ACCOUNT	Accountant
....				

JOIN and Sorting

SQL Review

- It is impossible to predict the sort order of a JOIN result
 - Unless ORDER BY is specified
 - Example: Sort the result of the previous query by job title and employee ID

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
  FROM employees e
 JOIN jobs j
    ON e.job_id = j.job_id
 WHERE e.job_id = 'FI_ACCOUNT'
 ORDER BY job_title, employee_id;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE
109	Daniel	Faviet	FI_ACCOUNT	Accountant
111	Ismael	Sciarra	FI_ACCOUNT	Accountant
....				

More than Two Tables

SQL Review

- There may be more than two tables in a JOIN
 - All tables must have a JOIN condition
 - The keywords JOIN and ON are repeated for each additional table

- Example: Include department name
 - department_name is a column in the departments table
 - The relationship between employees and departments is department_id

```
SELECT e.employee_id
      , e.first_name
      , e.last_name
      , e.job_id
      , j.job_title
      , d.department_name
  FROM employees e
  JOIN jobs j
  ON e.job_id = j.job_id
  JOIN departments d
  ON e.department_id = d.department_id
 ORDER BY job_title, employee_id;
```

More than Two Tables (continued)

SQL Review

- Partial result of the query on the previous slide:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	JOB_TITLE	DEPARTMENT_NAME
167	Amit	Banda	SA_REP	Sales Rep	Sales
168	Lisa	Ozer	SA_REP	Sales Rep	Sales
197	Kevin	Feeney	SH_CLERK	Shipping Clerk	Shipping
198	Donald	OConnell	SH_CLERK	Shipping Clerk	Shipping
199	Douglas	Grant	SH_CLERK	Shipping Clerk	Shipping
....					



Exercise 8.3: Working with Multi-Table Queries

45 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

Inserting New Records

Transactions

How to Test JDBC

Deleting and Updating Records

Working with Multiple Tables

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Manipulating data using Data Manipulation Language (DML):
 - INSERT
 - UPDATE
 - DELETE
- Executing DML commands with JDBC
- Managing transactions
 - ACID
 - Isolation levels
- Using transactions in testing
 - Use a TransactionManager
 - Start a transaction before each test
 - Rollback the transaction after each test

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 9: Working with a Data Access Object

Chapter Overview

In this chapter, we will explore:

- Debugging Data Access Object methods
- Using a Business Service
- Testing a Business Service

Chapter Concepts

Debugging Data Access Objects

Business Service

Testing a Business Service

Chapter Summary

Debugging Guidelines

- Read the error message
- Analyze the stack trace
 - Start at the top
 - Read downward until you find a call to a method that is part of your application
 - That is a good place to set a breakpoint
- Run the test that causes the error in Debug mode
 - Step through the code starting at the breakpoint
 - Examine local variable values

Responding to Bug Reports

- Read the error message to understand the error condition
- **Important!** Write a test that will **fail** because of the error
- Run the test to verify that you get the **Red** bar
- Use your debugging skills to find the solution for the error
- Run the test to verify that you get the **Green** bar
- You now have a test to ensure that the error has been corrected
- If the error should ever occur again, the test will inform you with a nice **Red** bar



30 min

Exercise 9.1: Debugging Data Access Object Methods

- Complete this exercise described in the Exercise Manual

Chapter Concepts

Debugging Data Access Objects

Business Service

Testing a Business Service

Chapter Summary

Business Service

- A Business Service defines methods that provide solutions for stated requirements
 - Often obtains and returns data from backend data sources
 - Databases
 - File systems
 - Etc.
- Will often have a dependency on one or more DAOs
 - A service method will often call on one or more DAO methods to perform database operations
- The service methods will make calls to the DAO methods
 - The service methods should be executed in a transaction
- Question: Who manages the transaction?
 - We will answer this question soon

Business Service Calls on DAO Methods

```
public class EmployeeManagementService implements EmployeeManagement {  
    private EmployeeDao dao;  
  
    public EmployeeManagementService(EmployeeDao dao) {  
        this.dao = dao;  
    }  
  
    @Override  
    public List<Employee> fetchEmployees() {  
        List<Employee> employees = dao.queryAllEmployees();  
        return employees;  
    }  
  
    @Override  
    public Employee updateEmployee(Employee employee) {  
        Employee employee = dao.updateEmployee(employee);  
        return employee;  
    }  
}
```

Chapter Concepts

Debugging Data Access Objects

Business Service

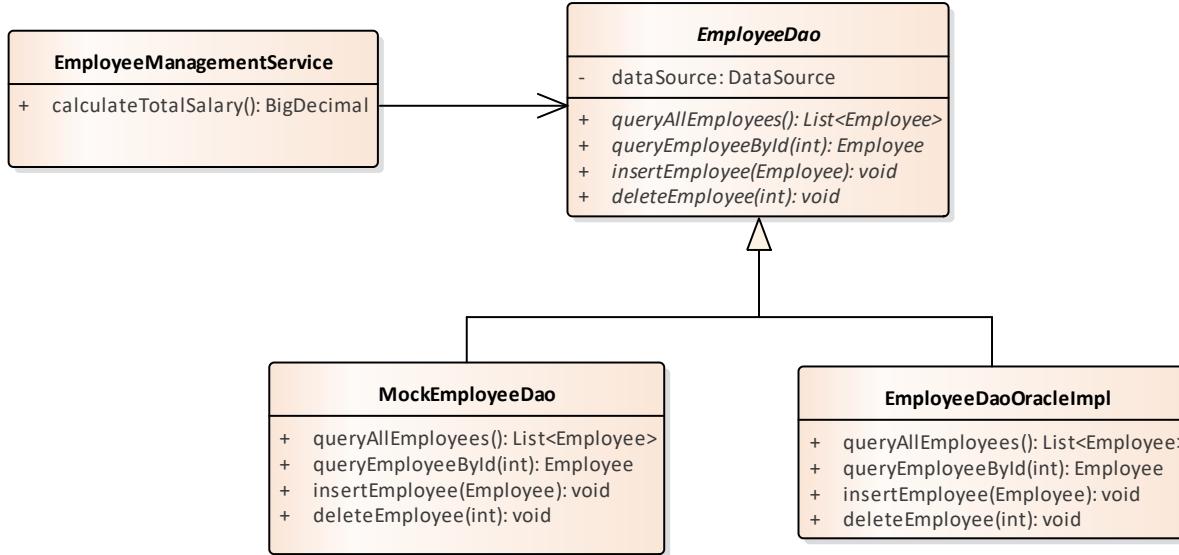
Testing a Business Service

Chapter Summary

Unit Testing a Business Service

- The EmployeeManagementService has a dependency on an EmployeeDao
- To write unit tests for the EmployeeManagementService, we do not want to use the EmployeeDaoOracleImpl
 - Instead, we will use a MockEmployeeDao
 - We want to focus on testing the EmployeeManagementService functionality, not the DAO functionality
 - We already have tests for the DAO functionality
- The MockEmployeeDao will inherit from EmployeeDao
 - The MockEmployeeDao will not interact with a database
 - It will return hard coded data

Testing a Business Service with a Mock DAO



Mock DAO

```
class MockDao extends EmployeeDao {  
    List<Employee> employees;  
  
    MockDao(DataSource ds) {  
        super(ds);  
        employees = new ArrayList<>();  
  
        employees.add(new Employee(7369, "SMITH", "CLERK", 7902,  
            LocalDate.parse("1980-12-17"), new BigDecimal("800.00"), null, 20));  
        employees.add(new Employee(7934, "MILLER", "CLERK", 7782,  
            LocalDate.parse("1982-01-23"), new BigDecimal("1300.00"), null, 10));  
    }  
  
    @Override  
    public List<Employee> queryAllEmployees() {  
        return employees;  
    }  
    ...  
}
```

Creating Mock Objects with Mockito

- Instead of writing the mock DAO class ourselves, we can use the Mockito framework
 - Mockito makes it easy to create mock objects and control their behavior

```
import org.mockito.*;  
class EmployeeManagementServiceTest {  
    @Mock EmployeeDao mockDao;  
    @InjectMocks EmployeeManagementService service;  
    @BeforeEach  
    void setUp() throws Exception {  
        MockitoAnnotations.openMocks(this);  
    }  
    @Test  
    void testQueryAllEmployees() {  
        List<Employee> expectedEmps = List.of(new PartTimeEmployee(7369, ...), new FullTimeEmployee(7934, ...));  
        Mockito.when(mockDao.queryAllEmployees())  
            .thenReturn(expectedEmps);  
        List<Employee> actualEmps = service.fetchEmployees();  
        assertEquals(expectedEmps, actualEmps);  
    }  
}
```

1. Mockito creates a mock DAO

Fields marked with `@Mock` are injected into the object marked with `@InjectMocks`

2. Mockito calls the `EmployeeManagementService` constructor, passing the mock DAO as an argument

3. Configure mock DAO to return expected employee list when its `queryAllEmployees()` method is called

4. Call service method

5. Verify service method handles the returned list correctly

Creating Mock Objects with Mockito (continued)

- With a mock DAO, it's easy to test conditions that are hard to reproduce with a real DAO
 - A DAO method call that returns an empty list or `null`
 - A DAO method call that throws an exception

```
@Test  
void testQueryAllEmployees_DaoReturnsEmptyList() {  
    Mockito.when(mockDao.queryAllEmployees())  
        .thenReturn(Collections.emptyList());  
  
    List<Employee> actualEmps = service.fetchEmployees();  
  
    assertEquals(Collections.emptyList(), actualEmps);  
}  
  
@Test  
void testQueryAllEmployees_DaoThrowsException() {  
    Mockito.when(mockDao.queryAllEmployees())  
        .thenThrow(new DatabaseException());  
  
    assertThrows(DatabaseException.class, () ->  
        service.fetchEmployees()  
);  
}
```

Configure mock DAO to return an empty list when its `queryAllEmployees()` method is called

Verify the service method handles the empty list correctly

Configure mock DAO to throw an exception when its `queryAllEmployees()` method is called

Verify the service method handles the exception correctly

Optional Exercise 9.2: Testing a Business Service with a Mock DAO



30 min

- Complete this exercise described in the Exercise Manual



Exercise 9.3: Putting It All Together

90 min

- Complete this exercise described in the Exercise Manual
- In this exercise, you will create a new project that will contain a Data Access Object that will define methods for queries, inserts, updates, and deletes on an Oracle database
- You will define tests for all of the DAO methods
- Of course, you will do this TDD style!
- The tests for the DML operations will be performed in a transaction that is rolled back after the test completes

Chapter Concepts

Debugging Data Access Objects

Business Service

Testing a Business Service

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Debugging Data Access Object methods
- Using a Business Service
- Testing a Business Service



Technology Immersion Program

Working with Relational Databases

Chapter 10: Advanced JDBC

Chapter Overview

In this chapter, we will explore:

- Calling stored procedures
- Managing one-to-many relationships
- Using the Proxy design pattern to manage transactions for a business service
- Testing the proxy by using Mockito
- The testing pyramid and the need for integration testing

Chapter Concepts

JDBC Code Review

Stored Procedures

One-to-Many Queries

Using a Service Proxy

Integration Tests

Chapter Summary



HANDS-ON
EXERCISE

Optional Exercise 10.1: JDBC Code Review

30 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

JDBC Code Review

Stored Procedures

One-to-Many Queries

Using a Service Proxy

Integration Tests

Chapter Summary

Stored Procedures

- Use CallableStatement to invoke stored procedures

```
@Override  
public String updateByProcedure(int deptNumber) {  
    String sql = "{CALL update_dept(?, ?)}";  
    String result = null;  
    Connection conn = getConnection();  
    try (CallableStatement stmt = conn.prepareCall(sql)) {  
        stmt.setInt(1, deptNumber);  
        stmt.registerOutParameter(2, java.sql.Types.VARCHAR);  
        stmt.executeUpdate();  
        result = stmt.getString(2);  
    } catch (SQLException e) {  
        logger.error("Cannot update dept by proc {}", sql, e);  
        throw new DatabaseException("Cannot update dept by  
proc " + sql, e);  
    }  
    return result;  
}
```

Note JDBC-specific escape syntax

Output parameters must be registered before execution

Output parameters retrieved from statement

```
CREATE OR REPLACE PROCEDURE update_dept  
(  
    parm_deptno IN dept.deptno%TYPE,  
    parm_dname OUT dept.dname%TYPE  
)  
...
```

Chapter Concepts

JDBC Code Review

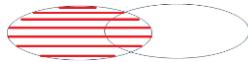
Stored Procedures

One-to-Many Queries

Using a Service Proxy

Integration Tests

Chapter Summary



Left Outer Join Example

SQL Review

- Left outer join returns all rows from the left table plus any matching rows in the right table
 - NULL values are added to columns when there is no match
 - The first table in the FROM clause is the left table
- Answers business questions like:
 - “I need all departments, including departments with no employees. For each department, I need all employees of the department.”

```
SELECT d.deptno, d.dname, d.loc,
       e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm
  FROM dept d
    LEFT OUTER JOIN emp e ON e.deptno = d.deptno
 ORDER BY d.deptno, e.empno;
```

DEPTNO	DNAME	LOC	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
12	30 SALES	CHICAGO	7698	BLAKE	MANAGER	7839	01-MAY-81	2850	(null)
13	30 SALES	CHICAGO	7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0
14	30 SALES	CHICAGO	7900	JAMES	CLERK	7698	03-DEC-81	950	(null)
15	40 OPERATIONS	BOSTON	(null)	(null)	(null)	(null)	(null)	(null)	(null)

Processing the ResultSet for an OuterJoin

- Processing the ResultSet returned from the Left Outer Join on the previous slide must be done carefully
- Every record in the ResultSet will contain the Department fields and fields for one Employee
- Only one Department object should be created for each unique department number

```
int previousDeptId = -1;
List<Department> departments = new ArrayList<>();
while (rs.next()) {
    List<Employee> emps;
    if (rs.getInt("deptno") != previousDeptId) {
        int departmentNumber = rs.getInt("deptno");
        String departmentName = rs.getString("dname");
        String departmentLocation = rs.getString("loc");
        emps = new ArrayList<>();
        department = new Department(departmentNumber,
                                      departmentName, departmentLocation, emps);
        departments.add(department);
        previousDeptId = departmentNumber;
    }
    int empNumber = rs.getInt("empno");
    if (!rs.wasNull()) {
        String empNumber = rs.getString("empno");
        ...
        Employee emp = new Employee(empNumber, ...);
        emps.add(emp);
    }
}
return departments;
```

Testing One-to-Many Query

- Assert the id of the Department is the expected value
- Assert that the departmentNumber of each Employee is the expected Department number

```
@Test  
void testQueryForDepartment() {  
    int id = 10;  
  
    Department department = dao.queryForDepartment(id);  
  
    assertNotNull(department);  
    assertEquals(id, department.getId());  
    for (Employee employee : department.getEmployees()) {  
        assertEquals(id, employee.getDeptNumber());  
    }  
}
```

A more thorough test would verify every property of the Department and all Employees



HANDS-ON
EXERCISE

Optional Exercise 10.2: One-to-Many Queries

45 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

JDBC Code Review

Stored Procedures

One-to-Many Queries

Using a Service Proxy

Integration Tests

Chapter Summary

Managing Transactions for Business Service Methods

- The Business Service defines methods that call on DAO methods
 - Should be executed inside a transaction
 - Can use a TransactionManager
- The Business Service methods could call on the TransactionManager
 - Start a transaction
 - Perform the actions for the business service method
 - Commit the transaction
 - Roll back the transaction if there is a problem
- This would require every Business Service method to perform these steps
- The Business Service then would have a dependency on the TransactionManager
 - And knowledge of database transactions is now in the Business Service

Managing Transactions

■ The Problem:

- The Business Service methods must execute in a transaction
- We can use a TransactionManager to start, commit, and roll back a transaction
- We don't want the Business Service to have a dependency on the TransactionManager

■ Solution:

- Find a Design Pattern that manages database transactions when Business Service methods are called by using a TransactionManager
- The Business Service will not have explicit knowledge of the database transaction or the TransactionManager

Structural Design Patterns

Adapter

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

- Decouple an abstraction from its implementation so that the two can vary independently.

Composite

- Compose objects into tree structure to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Façade

- Provide a unified interface to a set of interfaces in a system. Façade defines a higher-level interface that makes the subsystem easier to use.

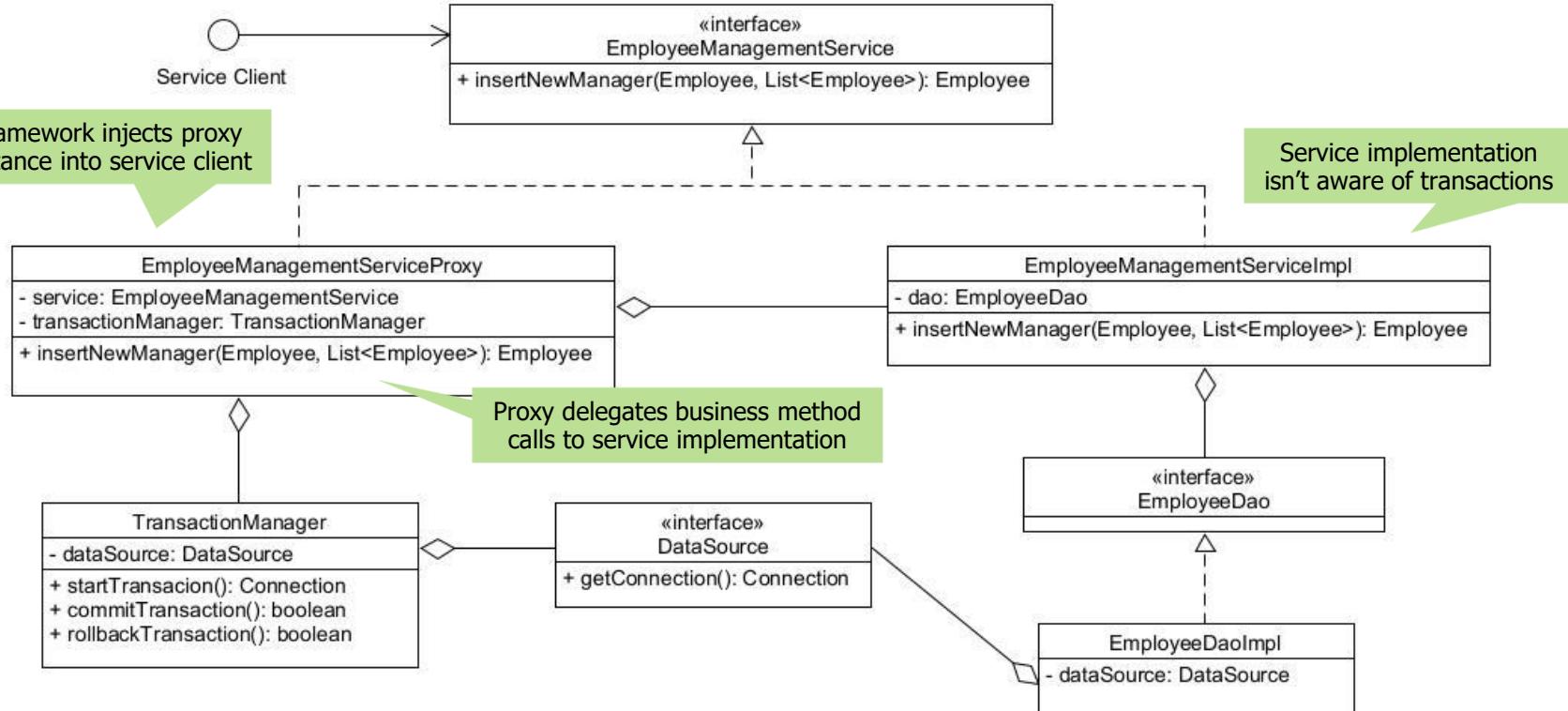
Proxy

- Provide a surrogate or placeholder for another object to control access to that object.

Proxy

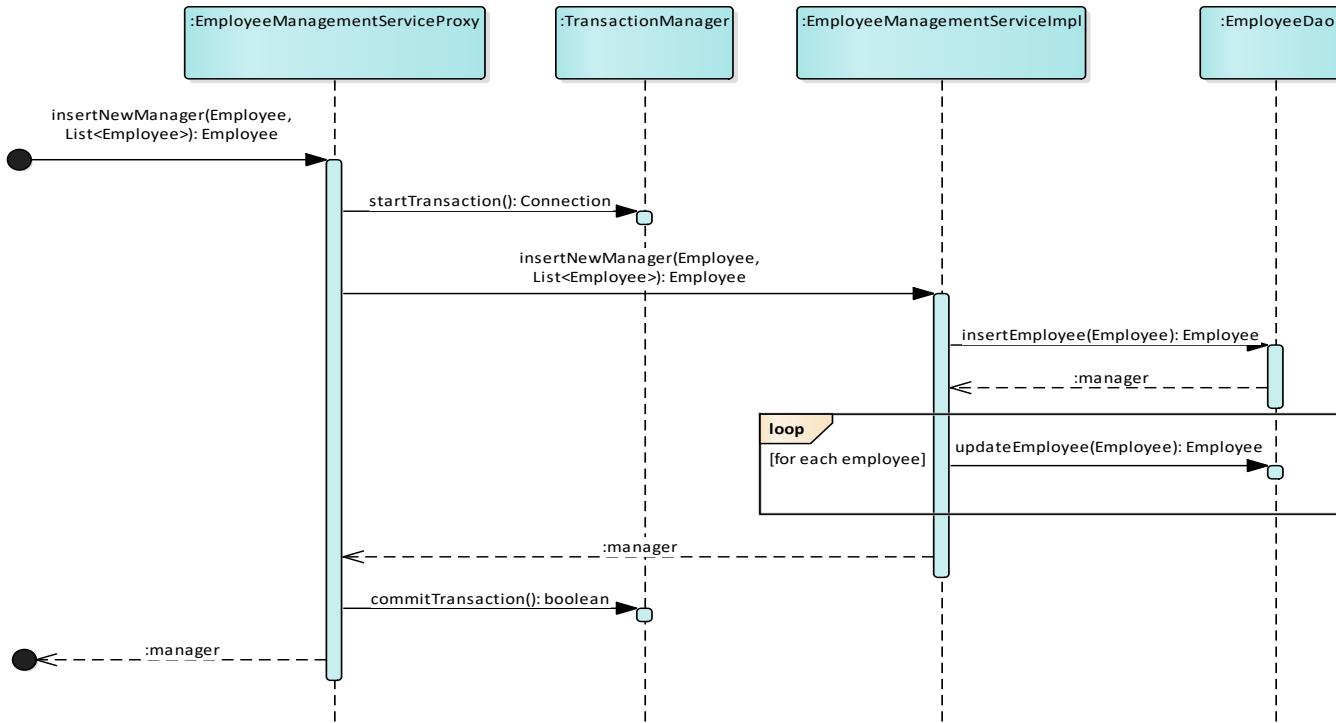
- Design Pattern: Proxy
- Problem: How to utilize a representative of another object to provide additional functionality such as database transactions, security, communications, or performance
- Solution: Define a proxy class that implements the same interface as the target object
 - Proxy will define additional functionality
 - Client calls methods of the proxy instead of the target object
 - Proxy method adds functionality (e.g., starts a transaction), then calls the target method
- Consequences:
 - Proxy has an opportunity to define pre- and post-processing
 - Proxy can determine if it will delegate the call to the original object
 - The functioning of the Proxy is transparent to the client

Service Proxy Class Diagram



Proxy

Scenario: Insert a new Employee in a transaction



Testing EmployeeManagementServiceProxy

- As shown in the sequence diagram, we want the service proxy to do the following:
 - Call startTransaction() **on the** TransactionManager
 - Call EmployeeManagementService methods that are part of the transaction
 - Call commitTransaction() **or** rollbackTransaction() **on** TransactionManager
- In testing, we want to verify the proxy called these methods in the correct order
- In these tests, we do NOT have to verify the functionality of the EmployeeManagementService or the TransactionManager
 - We already have tests for those classes
- We can use Mockito for this purpose
 - Mockito will create mock objects to replace EmployeeManagementService **and** TransactionManager
 - Mockito will inject those mock objects into the proxy being tested

Setting Up Mockito

- Mockito creates mock objects for the proxy's dependencies and injects them in the proxy
- When the method under test is called, the proxy calls methods of the two mock objects
- Mockito can verify the expected methods were called with the correct arguments

```
public class EmployeeManagementServiceProxy {  
    private EmployeeManagementService service;  
    private TransactionManager txnManager;  
  
    public EmployeeManagementServiceProxy(  
        EmployeeManagementService service,  
        TransactionManager txnManager) {  
        this.service = service;  
        this.txnManager = txnManager;  
    }  
}
```

Mockito creates mock objects for these two fields

Mockito injects fields marked with `@Mock` into the field marked with `@InjectMocks`

```
class EmployeeManagementServiceProxyTest {  
    @Mock private EmployeeManagementService mockService;  
    @Mock private TransactionManager mockTxnManager;  
    @InjectMocks private EmployeeManagementServiceProxy proxy;
```

```
@BeforeEach  
void setUp() {  
    MockitoAnnotations.openMocks(this);  
}
```

By default, Mockito calls the "biggest" constructor, passing the two mocks as arguments

Testing with Mockito

- These tests verify the EmployeeManagerServiceProxy calls methods of TransactionManager and EmployeeManagerService as expected
 - You can verify mock methods were called without regard to the order of calls, OR
 - You can verify mock methods were called in a specific order

```
import static org.mockito.Mockito.verify;

@Test
void testInsertEmployee() {
    Employee emp = new Employee(...);

    // Call the proxy method under test
    proxy.insertEmployee(emp);

    // Verify mock methods were called with
    // the correct arguments
    verify(mockTxnManager).beginTransaction();
    verify(mockService).insertEmployee(emp);
    verify(mockTxnManager).commitTransaction();
```

```
@Test
void testInsertEmployee() {
    Employee emp = new Employee(...);

    // Call the proxy method under test
    proxy.insertEmployee(emp);

    // Verify methods were called with the correct
    // arguments and in the proper order
    InOrder inOrder = Mockito.inOrder(mockTxnManager,
                                       mockService);
    inOrder.verify(mockTxnManager).beginTransaction();
    inOrder.verify(mockService).insertEmployee(emp);
    inOrder.verify(mockTxnManager).commitTransaction();
}
```

Optional Exercise 10.3: Testing a Service Proxy with Mockito



30 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

JDBC Code Review

Stored Procedures

One-to-Many Queries

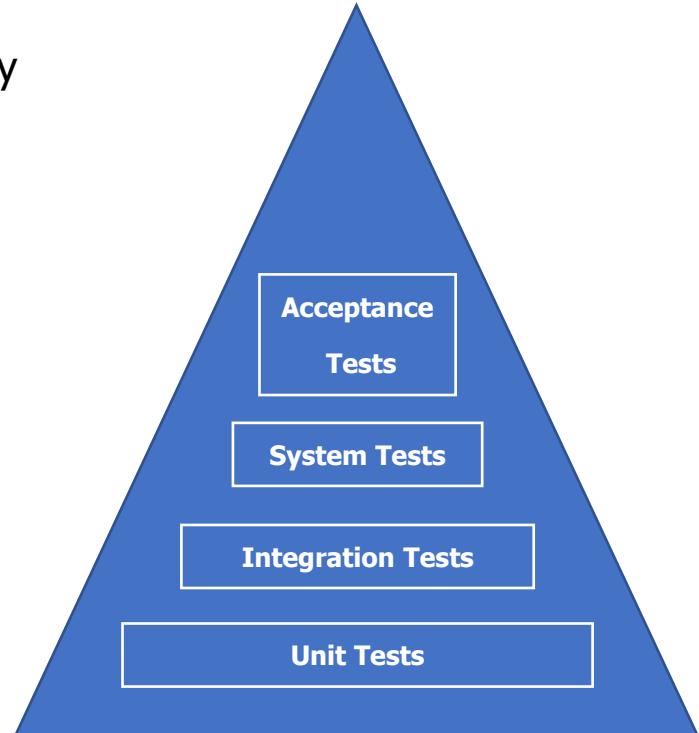
Using a Service Proxy

Integration Tests

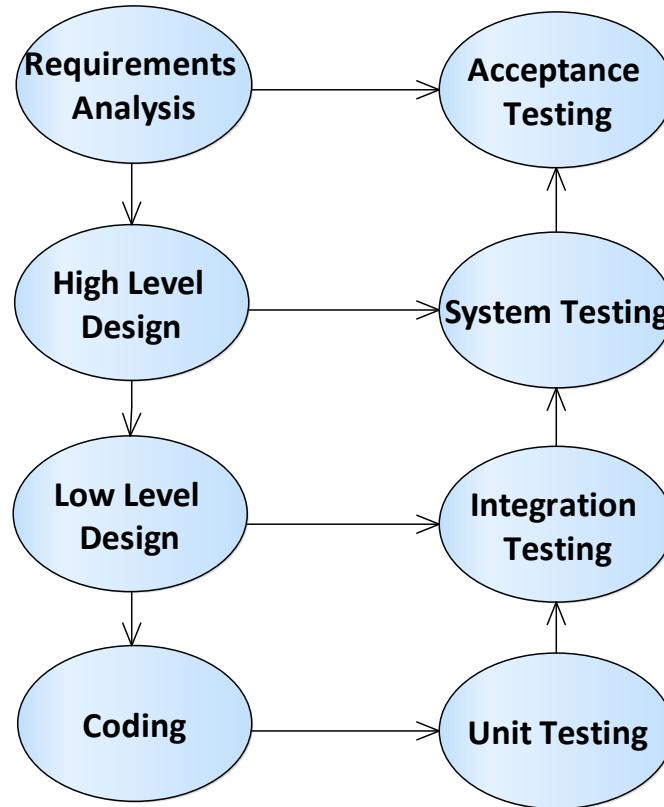
Chapter Summary

Integration Testing

- All the tests written so far have been unit tests
 - They verify one type of object is operating correctly
- Some integration tests are important
 - To verify that two objects work together correctly
- Typically, there will be many more unit tests than integration tests



Activities Related to Testing



Integration vs. Unit Testing

Integration Testing	Unit Testing
Focuses on the interaction between units	Focuses on one unit (class)
Units are combined and tested as a group	One unit is tested
Real object instances are used in tests	Dependencies are usually mocked in tests
Verifies classes communicate successfully	Verifies methods in a single class perform correctly

Integration Test for the Service and DAO

- Integration tests will use the real (no mocks) service and DAO objects
- An integration test will call on an EmploymentManagementServiceImpl object
- The EmploymentManagementServiceImpl object will call on the EmployeeDaoOracleImpl DAO
- The EmploymentManagementServiceImpl does not manage a transaction explicitly
 - The test class can start a transaction before each test
 - The test class can roll back the transaction after each test
 - This ensures that the database will not be altered by our tests

Initializing and Cleaning Up Integration Tests

- The `@BeforeEach` method creates the objects used in the integration tests and starts a transaction
- The `@AfterEach` method rolls back the transaction and shuts down the `DataSource` which closes the database Connection

```
@BeforeEach
void setUp() throws Exception {
    dataSource = new SimpleDataSource();
    dao = new EmployeeDaoOracleImpl(dataSource);
    service = new EmployeeManagementServiceImpl(dao);
    transactionManager =
        new TransactionManager(dataSource);

    // Start the transaction
    connection = dataSource.getConnection();
    transactionManager.startTransaction();

    dbTestUtils = new DbTestUtils(connection);
    jdbcTemplate = dbTestUtils.initJdbcTemplate();
}
```

```
@AfterEach
void tearDown() throws Exception {
    // Rollback the transaction
    transactionManager.rollbackTransaction();

    // Shutdown the DataSource
    // This closes the database Connection
    dataSource.shutdown();
}
```

Optional Exercise 10.4: Writing an Integration Test for the Service and DAO



30 min

- Complete this exercise described in the Exercise Manual

Chapter Concepts

JDBC Code Review

Stored Procedures

One-to-Many Queries

Using a Service Proxy

Integration Tests

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Calling stored procedures
- Managing one-to-many relationships
- Using the Proxy design pattern to manage transactions for a business service
- Testing the proxy by using Mockito
- The testing pyramid and the need for integration testing



Top 20 Replies by Programmers when their programs don't work...

20. That's weird...
19. It's never done that before.
18. It worked yesterday.
17. How is that possible?
16. It must be a hardware problem.
15. What did you type in wrong to get it to crash?
14. There has to be something funky in your data.
13. I haven't touched that module in weeks!
12. You must have the wrong version.
11. It's just some unlucky coincidence.
10. I can't test everything!
9. THIS can't be the source of THAT.
8. It works, but it hasn't been tested.
7. Somebody must have changed my code.
6. Did you check for a virus on your system?
5. Even though it doesn't work, how does it feel?
4. You can't use that version on your system.
3. Why do you want to do it that way?
2. Where were you when the program blew up?
1. It works on my machine.

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 11: Aggregating Information

Chapter Overview

In this chapter, we will explore:

- Reporting about groups of data
 - Aggregate functions
- Defining the groups
 - The GROUP BY and HAVING clauses
- JOINING issues
- Working with subqueries

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

Aggregate Functions

- Aggregate functions return one result per group of rows, rather than one result per row
 - For simplicity of discussion, assume one group is retrieved
 - A SELECT statement can retrieve multiple groups by utilizing the GROUP BY clause, which will be discussed later
- The common aggregate functions are:
 - COUNT
 - SUM
 - AVG
 - MAX
 - MIN
- **All aggregate functions ignore NULL values**
 - Except COUNT (*)

Aggregate Function: COUNT

- COUNT always returns a number
 - Even if no rows are counted, COUNT will return 0
- COUNT can be used several ways
 - COUNT (*) returns the number of rows in the group
 - COUNT(some_column) returns the number of non-NULL values in the group
 - COUNT(DISTINCT some_column) returns the number of distinct (unique) values

COUNT: Example

Given the following set

- How many rows are there?
- How many rows have a minimum salary?
- How many distinct minimum salary values are there?

```
SELECT job_title, min_salary
FROM jobs
WHERE min_salary < 5000 ORDER BY min_salary;
```

JOB_TITLE	MIN_SALARY
Stock Clerk	2000
Purchasing Clerk	2500
Shipping Clerk	
Administration Assistant	3000
Programmer	4000
Human Resources Representative	4000
Marketing Representative	4000
Accountant	4200
Public Accountant	4200
Public Relations Representative	4500

COUNT: Example (continued)

Answer:

```
SELECT COUNT(*), COUNT(min_salary), COUNT(DISTINCT min_salary)
FROM jobs
WHERE min_salary < 5000;
```

COUNT(*)	COUNT(MIN_SALARY)	COUNT(DISTINCTMIN_SALARY)
----------	-------------------	---------------------------

-----	-----	-----
10	9	6

Question: How many employees have commission, how many do not, and what is the total?

```
SELECT COUNT(commission_pct) "Have Commission",
       COUNT(*) - COUNT(commission_pct) "Do Not Have Commission",
       COUNT(*) "Total Emps"
FROM employees;
```

Have Commission	Do Not Have Commission	Total Emps
-----------------	------------------------	------------

-----	-----	-----
35	72	107

Aggregate Functions: SUM and AVG

- Remember, the aggregate functions ignore NULL values
 - `SUM(some_column)` totals up the non-NULL values
 - `AVG(some_column)` averages the non-NULL values
 - `DISTINCT` can also be used
 - `SUM(DISTINCT some_column)`
 - `AVG(DISTINCT some_column)`
- Be sure that this value is what you intend to return!
 - You may prefer to treat NULL as 0 by using `COALESCE`

SUM and AVG Examples

- What are the sum and average of commission percent values?

```
SELECT SUM(commission_pct) AS sum,
       COUNT(commission_pct) AS "COUNT NOT NULL",
       ROUND(AVG(commission_pct), 6) AS avg,
       ROUND(SUM(commission_pct) / COUNT(commission_pct), 6) AS "AVG BY CALC",
       COUNT(*) AS count,
       ROUND(AVG(COALESCE(commission_pct, 0)), 6) AS "AVG (NULL=0)",
       ROUND(SUM(commission_pct) / COUNT(*), 6) AS "AVG BY CALC (NULL=0)"
FROM employees;
```

SUM	COUNT	NOT NULL	AVG	AVG BY CALC	COUNT	AVG (NULL=0)	AVG BY CALC (NULL=0)
7.8	35	.222857	.222857	.222857	107	.072897	.072897

Aggregate Functions: MAX and MIN

- These functions return the largest and smallest values of a column in a set of data
- What are the largest and smallest commission percent values?
 - Notice that NULL values do not show as either MAX or MIN

```
SELECT MAX(commission_pct), MIN(commission_pct) FROM employees;  
  
MAX(COMMISSION_PCT) MIN(COMMISSION_PCT)  
-----  
          .4            .1
```

- Non-numeric data
 - What employee's last name appears at the end of a sorted listing?
 - What is the oldest hire date for employees on file?

```
SELECT MAX(last_name) FROM employees;  
MAX(LAST_NAME)  
-----  
zlotkey
```

```
SELECT MIN(hire_date) FROM employees;  
MIN(HIRE_DA  
-----  
17-JUN-1987
```

Mixing Single Row (Scalar) and Aggregate Values

- When reporting an aggregate value about a group, we cannot also return a scalar value
- Given this set, we cannot return an aggregate value AND a single row value

JOB_TITLE	MIN_SALARY
Stock Clerk	2000
Purchasing Clerk	2500
Shipping Clerk	
Administration Assistant	3000
Programmer	4000
Human Resources Representative	4000
Marketing Representative	4000
Accountant	4200
Public Accountant	4200
Public Relations Representative	4500

```
SELECT job_title, MAX(min_salary) FROM jobs  
WHERE min_salary < 5000;  
SELECT job_title, MAX(min_salary) FROM jobs  
*  
ERROR at line 1:  
ORA-00937: not a single-group group function
```



Exercise 11.1: Using the Aggregate Functions

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

Specifying the Groups

- The sets of data we have been aggregating have been tables
 - Or a result set of tables filtered by the WHERE clause
- Suppose we wanted to describe aggregates for different groups
 - The average minimum and maximum salaries for different jobs
 - Defined by the first two characters of the job_id column
- We could run a series of aggregate queries against the jobs table
 - Using the WHERE clause to filter out one group at a time

Groups the Hard Way

- This approach works, but is making several assumptions
 - That we know all of the `job_id` strings (we could query them first)
 - That we have the patience to build and execute *many* statements!

```
SELECT AVG(min_salary), AVG(max_salary), COUNT(*)  
FROM jobs  
WHERE SUBSTR(job_id, 1, 2) = 'AD';
```

AVG (MIN_SALARY)	AVG (MAX_SALARY)	COUNT (*)
12666.6667	25333.3333	3

```
SELECT AVG(min_salary), AVG(max_salary), COUNT(*)  
FROM jobs  
WHERE SUBSTR(job_id, 1, 2) = 'FI';
```

AVG (MIN_SALARY)	AVG (MAX_SALARY)	COUNT (*)
6200	12500	2

Groups the Easy Way

- Fortunately, SQL allows us to specify GROUPS in the SELECT statement
 - The GROUP BY clause
- And we can filter out groups we decide not to include in final result set
 - The HAVING clause
 - Discussed in a few minutes
- Remember the overall structure of the SELECT statement

```
SELECT    column or expression, column or expression ...
FROM        table
WHERE      condition 1 AND/OR condition 2 ...
GROUP BY column or expression, column or expression ...
HAVING   condition 1 AND/OR condition 2 ...
ORDER BY column or expression or column alias or position, ...
```

GROUP BY Clause

- All columns in the SELECT clause must be in the GROUP BY clause or must be part of an aggregate function
- The aggregate will produce one row per group
 - NULL values in the GROUP BY column will be grouped into a single group
- The condition(s) in the HAVING clause are then applied to the grouped sets
 - And are accepted or filtered out
- Remember, if the result set sequence is important, specify an ORDER BY clause

GROUP BY Example

- What is the highest and lowest employee id in each department?

```
SELECT department_id, MIN(employee_id), MAX(employee_id)
FROM employees
WHERE department_id > 10
GROUP BY department_id
ORDER BY department_id;
```

DEPARTMENT_ID	MIN(EMPLOYEE_ID)	MAX(EMPLOYEE_ID)
20	201	202
30	114	119
40	203	203
50	120	199
60	103	107
70	204	204
80	145	179
90	100	102
100	108	113
110	205	206

GROUP BY Example (continued)

- Add the count of how many employees are in each department and list the largest departments first and by department number descending within the employee count
 - Place employee 178 at the bottom

```
SELECT department_id, COUNT(*), MIN(employee_id), MAX(employee_id)
FROM employees
GROUP BY department_id
ORDER BY COUNT(*) DESC, department_id DESC NULLS LAST;
```

DEPARTMENT_ID	COUNT(*)	MIN(EMPLOYEE_ID)	MAX(EMPLOYEE_ID)
50	45	120	199
80	34	145	179
100	6	108	113
30	6	114	119
60	5	103	107
90	3	100	102
110	2	205	206
20	2	201	202
70	1	204	204
40	1	203	203
10	1	200	200
	1	178	178

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

The HAVING Clause

- The WHERE clause filters the rows selected from the table(s)
- The HAVING clause filters groups once the GROUP BY has completed
- The conditions are constructed using the same comparison operators as the WHERE clause
- The HAVING clause should only reference aggregates
 - Data that is NOT known until this time

HAVING Clause Example

- Restrict the reporting about departments to those which have at least five employees

```
SELECT department_id, COUNT(*), MIN(employee_id), MAX(employee_id)
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC, department_id DESC NULLS LAST;
```

DEPARTMENT_ID	COUNT(*)	MIN(EMPLOYEE_ID)	MAX(EMPLOYEE_ID)
50	45	120	199
80	34	145	179
100	6	108	113
30	6	114	119

Invalid WHERE Clause

- Can we filter out those departments with few employees in the WHERE clause?
 - Why or why not?

```
SELECT department_id, COUNT(*), MIN(employee_id), MAX(employee_id)
FROM employees
WHERE COUNT(*) > 5
GROUP BY department_id
ORDER BY COUNT(*) DESC, department_id DESC NULLS LAST;

WHERE COUNT(*) > 5
*
ERROR at line 3:
ORA-00934: group function is not allowed here
```

- We cannot because the data value is not yet available
 - Aggregate functions are not allowed in the WHERE clause

A Legal, But Inefficient HAVING Clause

- Suppose we wanted to restrict the list to department numbers less than 100
- The following query returns the correct result

```
SELECT department_id, COUNT(*), MIN(employee_id), MAX(employee_id)
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5
AND department_id <> 100
ORDER BY COUNT(*) DESC, department_id DESC NULLS LAST;
```

DEPARTMENT_ID	COUNT(*)	MIN(EMPLOYEE_ID)	MAX(EMPLOYEE_ID)
50	45	120	199
80	34	145	179
30	6	114	119

Same Answer, Only Quicker

- The department_id is known at the scalar level
 - And can be filtered in the WHERE clause

```
SELECT department_id, COUNT(*), MIN(employee_id), MAX(employee_id)
FROM employees
WHERE department_id <> 100
GROUP BY department_id
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC, department_id DESC NULLS LAST;
```

DEPARTMENT_ID	COUNT(*)	MIN(EMPLOYEE_ID)	MAX(EMPLOYEE_ID)
50	45	120	199
80	34	145	179
30	6	114	119

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

Query Problems Can Be Masked by Aggregates

- The result set being passed into aggregate processing could be coming from a JOIN
 - The JOIN may be syntactically correct
 - But not what you intended
 - Since the aggregate is a number, it could seem reasonable
 - But might lead to the wrong business decision
- Things to bear in mind
 - 1:M joins may result in column values being duplicated in the pre-aggregation results set
 - Outer joins introduce NULLs, which may not be handled as you expect
- **ALWAYS** make certain that the JOIN is complete first, then add the GROUP BY functionality



Exercise 11.2: GROUP BY and HAVING

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

Simple Subqueries

- A subquery is a `SELECT` statement that occurs inside a condition of another `SELECT` statement
 - Each row of the parent statement is compared with the result of the subquery
 - If the comparison fails, the row is rejected
- The subquery is the same as the regular `SELECT` statement without the `ORDER BY` clause
- A subquery can have its own subqueries
- Normally used in the `WHERE` clause of the parent `SELECT` statement
- Common comparison operators for a subquery
 - Relational operators (`=, <>, >, >=, <, <=`)
 - `IN` and `NOT IN`

Simple Subquery Using a Relational Operator

- Subquery must return a single value
- Example:
 - Find employees who have the maximum salary

```
SELECT e.last_name, e.first_name, e.salary
FROM employees e
WHERE e.salary = (
    SELECT MAX(e2.salary)
    FROM employees e2
) ;
```

Simple Subquery Using an IN Operator

- Subquery returns a list of valid values

- Example:

- Select the department_id and department_name for all departments being managed by any manager who is also managing employees named Smith
- Employee name is in the employees table
 - The employees subquery is expected to return multiple manager ids
 - Use the IN operator

```
SELECT department_id, department_name
FROM departments
WHERE manager_id IN (
    SELECT manager_id
    FROM employees
    WHERE last_name = 'Smith'
);
```

- NOT IN can replace the IN operator in this query to get departments for all other managers

Correlated Subquery

- A simple subquery is evaluated once, and its result is compared with each row of the parent query
- A correlated subquery is evaluated and compared once for each row of the parent table
 - Necessary because a correlated subquery uses data from the parent query which is expected to change with every row
 - Powerful because the decision is based on data in the parent query and subquery
- Syntax:
 - Same as a simple subquery, but references a column of the parent query
 - Reference is made by specifying a column from one of the tables of the parent query

EXISTS Operator

- Allows testing for existence rather than for specific values
 - Evaluates to TRUE if the subquery returns at least one row; FALSE otherwise
 - Efficient because the subquery stops after finding the first row
- Syntax:

WHERE [NOT] EXISTS subquery

- Does not compare the results of the subquery with anything
- NOT EXISTS
 - Evaluates to TRUE if the subquery returns no rows
 - Not efficient because the subquery must check all rows to complete

Correlated Subquery Example

- Find departments that do not have any employees
 - For each department
 - Find all employees for this department (correlate on `department_id`)

```
SELECT d.department_name
FROM departments d
WHERE NOT EXISTS (
    SELECT 1
    FROM employees e
    WHERE e.department_id = d.department_id
) ;
```

- Because `EXISTS` tests for existence, the data selected by the subquery are not used
 - “1” is used as a dummy value because the `SELECT` list cannot be empty



HANDS-ON
EXERCISE

20 min

Exercise 11.3: Using Subqueries

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Aggregate Functions

The GROUP BY Clause

The HAVING Clause

JOINing Issues

Subqueries

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Reporting about groups of data
 - Aggregate functions
- Defining the groups
 - The GROUP BY and HAVING clauses
- JOINING issues
- Working with subqueries

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 12: Set Operators

Chapter Overview

In this chapter, we will explore:

■ Set operators

- UNION, UNION ALL
- INTERSECT
- MINUS

Chapter Concepts

Set Operators

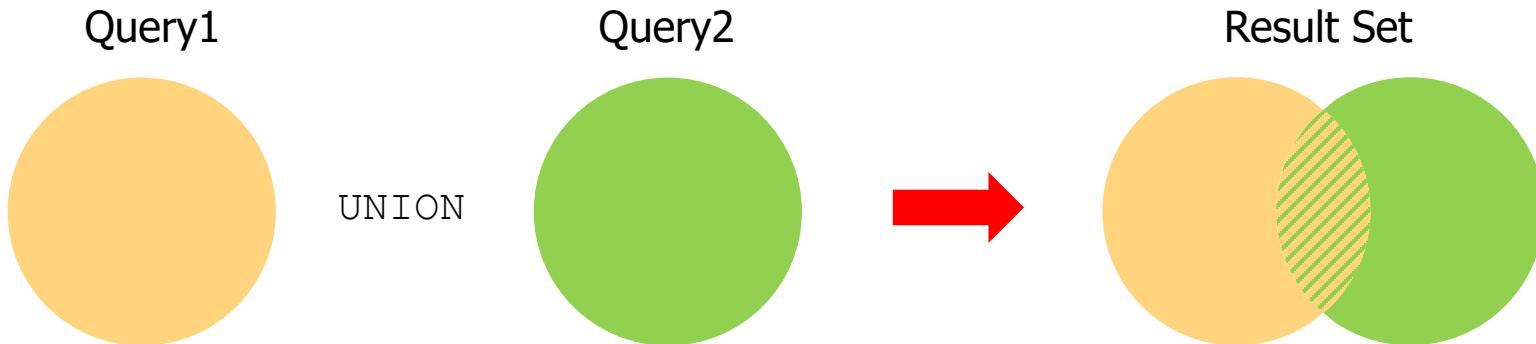
Chapter Summary

Set Operators

- Set operators specify interactions between whole tables or query results
 - Each set must be the same
 - The same number of columns
 - And the columns be the same datatype
 - Or be implicitly convertible
 - The first `SELECT` list is used for the column headings
 - Only one `ORDER BY` clause can be used
 - Applies to the entire result set
- Set operators are specified between the description of sets
 - Sets are nothing more than queries

UNION

- Returns the distinct rows from the combination of Query1 and Query2
 - UNION does an automatic distinct across the result of the two queries
 - The results are *NOT* implicitly sorted



UNION Example

ACTORS

FIRST_NAME	LAST_NAME
Priyanka	Chopra
Johnny	Depp
Tom	Hanks
Peter	O'Toole
Jing	Tian

OSCAR_WINNERS

FIRST_NAME	LAST_NAME
Tom	Hanks
Neil	Jordan
Ang	Lee
Peter	O'Toole

- List all the people once

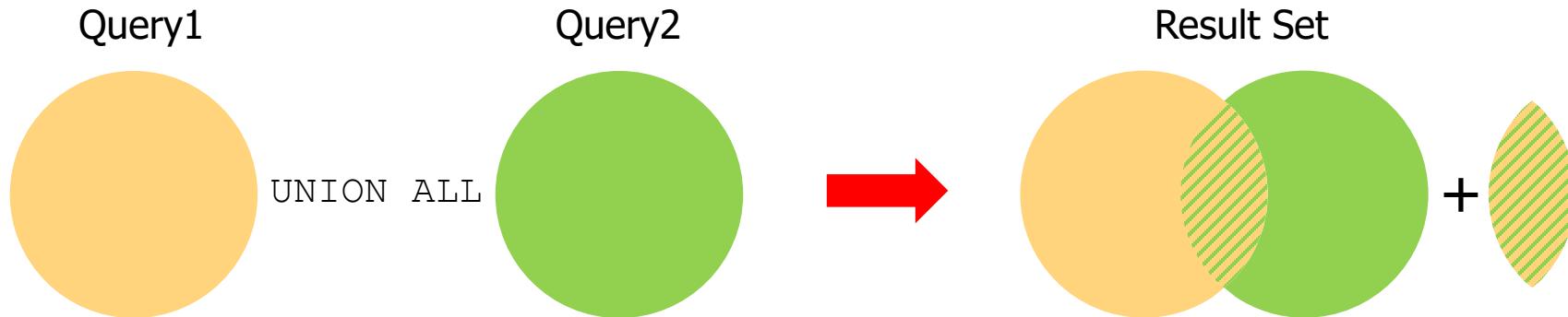
```
SELECT first_name, last_name FROM actors
UNION
SELECT first_name, last_name FROM oscar_winners
ORDER BY last_name, first_name;
```

FIRST_NAME	LAST_NAME
Priyanka	Chopra
Johnny	Depp
Tom	Hanks
Neil	Jordan
Ang	Lee
Peter	O'Toole
Jing	Tian

7 rows selected.

UNION ALL

- Returns ALL rows from both Query1 and Query2
 - UNION ALL is much more efficient than UNION
 - If you know that there will not be any duplicates, choose UNION ALL



UNION ALL Example

ACTORS

FIRST_NAME	LAST_NAME
Priyanka	Chopra
Johnny	Depp
Tom	Hanks
Peter	O'Toole
Jing	Tian

OSCAR_WINNERS

FIRST_NAME	LAST_NAME
Tom	Hanks
Neil	Jordan
Ang	Lee
Peter	O'Toole

List all the people as many times as they appear

```
SELECT first_name, last_name FROM actors
UNION ALL
SELECT first_name, last_name FROM oscar_winners
ORDER BY last_name DESC, first_name;
```

FIRST_NAME	LAST_NAME
-----	-----
Jing	Tian
Peter	O'Toole
Peter	O'Toole
Ang	Lee
Neil	Jordan
Tom	Hanks
Tom	Hanks
Johnny	Depp
Priyanka	Chopra

9 rows selected.

INTERSECT

- Returns rows from Query1 and Query2 If they are the same



- Hint: Under some conditions, `INTERSECTS` may be written as `JOINS`
 - If they can, the `JOIN` is usually more efficient

INTERSECT Example

ACTORS

FIRST_NAME	LAST_NAME
Priyanka	Chopra
Johnny	Depp
Tom	Hanks
Peter	O'Toole
Jing	Tian

OSCAR_WINNERS

FIRST_NAME	LAST_NAME
Tom	Hanks
Neil	Jordan
Ang	Lee
Peter	O'Toole

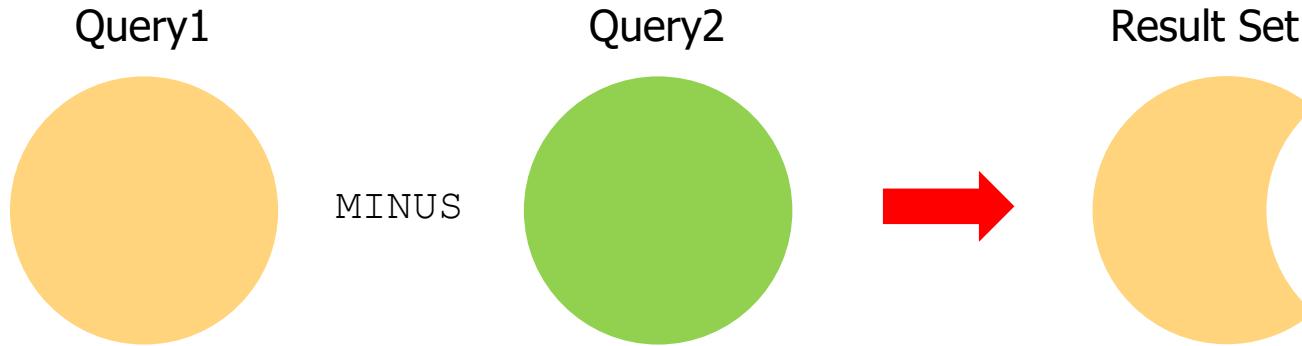
- List all the actors who are Oscar winners

```
SELECT first_name, last_name FROM actors
INTERSECT
SELECT first_name, last_name FROM oscar_winners
ORDER BY first_name;
```

FIRST_NAME	LAST_NAME
Peter	O'Toole
Tom	Hanks

MINUS

- Returns rows from Query1 that are NOT in Query2



- In the SQL standard, this is implemented as EXCEPT

MINUS Example

ACTORS

FIRST_NAME	LAST_NAME
Priyanka	Chopra
Johnny	Depp
Tom	Hanks
Peter	O'Toole
Jing	Tian

OSCAR_WINNERS

FIRST_NAME	LAST_NAME
Tom	Hanks
Neil	Jordan
Ang	Lee
Peter	O'Toole

- List all the actors who are not Oscar winners

```
SELECT first_name, last_name FROM actors  
MINUS  
SELECT first_name, last_name FROM oscar_winners  
ORDER BY last_name;
```

FIRST_NAME	LAST_NAME
-----	-----
Priyanka	Chopra
Johnny	Depp
Jing	Tian

SET Operators: More than One Query

- The construction can be:

query1 set_operator query2 set_operator query3 ...

- The grouping will be sequence
 - Unless forced by parentheses ()
- No matter how many operators, there can only be one ORDER BY



HANDS-ON
EXERCISE

10 min

Exercise 12.1: Set Operators

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Set Operators

Chapter Summary

Chapter Summary

In this chapter, we have explored:

■ Set operators

- UNION, UNION ALL
- INTERSECT
- MINUS

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 13: Programming with PL/SQL

Chapter Overview

In this chapter, we will explore:

- Working with PL/SQL
- Declaring, assigning, and using scalar variables
- Defining and using conditional, iterative, and sequential control
- Utilizing error handling and writing exception handlers
- Processing result sets with cursors
- Improving cursor processing with FOR-LOOP cursors
- Improving update and delete performance by using cursors

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

PL/SQL Example

Requirement:

- Employee William Smith (171) wishes to become an IT Programmer
- In our organization, we are limited to a maximum of 5 programmers
 - If present number is less than the maximum, promote William
 - Otherwise, add him to a waiting list for that position

```
DECLARE
    no_of_employees          NUMBER(2);
    max_employees      CONSTANT NUMBER(1) := 5;
BEGIN
    SELECT COUNT(*)
    INTO no_of_employees
    FROM employees
    WHERE job_id = 'IT_PROG';

    IF no_of_employees < max_employees THEN
        UPDATE employees
        SET job_id = 'IT_PROG'
        WHERE employee_id = 171;
    ELSE
        INSERT INTO waiting_list (
            employee_id,
            job_id
        )
        VALUES (171, 'IT_PROG');
    END IF;
END;
/
```

PL/SQL Basics

- PL/SQL stands for Procedural Language (PL) extensions to Structured Query Language (SQL)
- Developed by Oracle to be their standard programming language
- Each PL/SQL statement must end with a semicolon
 - Because an `IF` statement is not complete until the `END IF`, a semicolon is required after `END IF` but not allowed after `THEN`
- Similarly, because a block is not complete until the `END` key word, a semicolon is required after `END` but not allowed after `DECLARE` or `BEGIN`
- In PL/SQL, a semicolon does not execute the block (unlike in SQL)
 - A slash (/) is required to execute the block
 - The rest of the course notes will not include a slash

PL/SQL Basics (continued)

- The code can be stored in the database
 - Procedures and functions
 - Packages
 - Triggers
 - Can pass parameters
- Provides exception handling
 - Simplifies coding of checking for errors and their resolution
- A PL/SQL program uses embedded SQL statements to access the database tables
 - Similar to SQL statements embedded in other procedural languages

The Structure of PL/SQL

- Each PL/SQL block is executed as a statement
 - DECLARE ... BEGIN ... EXCEPTION ... END
 - With other statements embedded
- The DECLARATION section contains declarations
 - Variables, constants, exceptions, cursors, etc.
 - The declaration section is optional
- The EXECUTION section contains executable statements
 - Each block must have at least one executable statement
 - A mandatory section
- The EXCEPTION section contains executable statements for handling errors
 - This section is optional

Anonymous Block

DECLARE

Declaration Section

BEGIN

Execution Section

EXCEPTION

Exception Section

END;

Variables

■ Variables are declared in the declaration section

```
var_name [CONSTANT] datatype | table_name.column_name%TYPE [:= expression];
```

■ %TYPE

- “Anchored declaration” provides the datatype of a variable based on a database column
- Don’t need to change code if column definition changes

```
salary employees.salary%TYPE;
```

salary is NUMBER(8, 2) based on the employees column definition

■ Assignment operator (:=) is used to assign values to variables

- Initialize in the declaration section

```
max_students NUMBER(2) := 5;
```

- Assign in the executable section

```
salary := salary * 1.10;
```

■ Use the optional CONSTANT keyword if the value of the variable should never be changed

```
max_employees CONSTANT NUMBER(1) := 5;
```

DBMS_OUTPUT Package

Display variables using DBMS_OUTPUT package

- Example:**

```
DBMS_OUTPUT.PUT_LINE ('salary: ' || salary);
```

- Use SET SERVEROUTPUT ON command before executing the procedure**

Important procedures

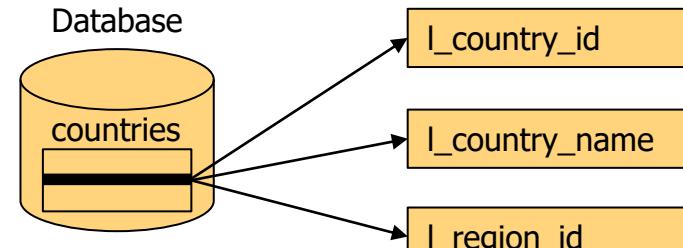
- PUT_LINE prints a whole line, including an end of line**
- PUT prints a partial line**
- NEW_LINE prints an end of line and may be used to end a line started with PUT**

Working with Variables

- Example: declare variables based on the countries table columns

```
DECLARE
    l_country_id    countries.country_id%TYPE;
    l_country_name  countries.country_name%TYPE;
    l_region_id     countries.region_id%TYPE;
BEGIN
    SELECT country_id, country_name, region_id
    INTO   l_country_id, l_country_name, l_region_id
    FROM   countries
    WHERE  region_id = 2
    AND    ROWNUM <= 1;

    -- Do something with the data
END;
```



- ROUNUM indicates the order of selected rows
 - Can be used to restrict the SELECT INTO to return one row
 - SELECT... INTO must have exactly one row (0 or more than 1 will cause an error)

Table Record Variables

- Combines related variables so that they can be manipulated as a unit

```
record_name      table_name%ROWTYPE;
```

- Example: declare a record based on the countries table row

```
country_rec      countries%ROWTYPE;
```

- country_rec:

country_id	country_name	region_id

- Select a row into the record type variable

```
SELECT *  
INTO   country_rec  
FROM   countries  
WHERE  region_id = 2  
AND    ROWNUM <= 1;
```

- Add a new country to the region

```
country_rec.country_id := 'PE';  
country_rec.country_name := 'PERU';  
  
INSERT INTO countries  
VALUES country_rec;
```

Naming Variables

- Variables are often named with a prefix or suffix:

- `c_country_id` (a type-based prefix, indicates it is character data)
- `l_country_id` (a scope-based prefix, indicates it is a local variable)
- Avoids clashes with column names, consider this (pointless) code:

```
DECLARE
    country_id    countries.country_id%TYPE := 'BR';
    country_name  countries.country_name%TYPE;
    region_id     countries.region_id%TYPE;
BEGIN
    SELECT country_id, country_name, region_id
    INTO   country_id, country_name, region_id
    FROM   countries
    WHERE  region_id = 2
    AND    country_id = country_id
    AND    ROWNUM <= 1;

    DBMS_OUTPUT.PUT_LINE(country_id);
END;
```

No confusion here because only variables can be the target of `SELECT... INTO`

One of these is meant to be the variable, but the column name is favored

Prints AR, not BR

Naming Variables (continued)

- Variables can always be qualified by the name of the enclosing block
 - More flexible and readable than a naming convention
 - Safer, in the (unlikely) event that a column called `l_country_id` is added to the table
- But, aren't these anonymous blocks?!
- Even anonymous blocks can have a (temporary) block name
 - Not stored, discarded after execution
- This course contains both styles

```
<<country_check>> Block name
DECLARE
    country_id    countries.country_id%TYPE := 'BR';
    country_name  countries.country_name%TYPE;
    region_id     countries.region_id%TYPE;
BEGIN
    SELECT c.country_id, c.country_name, c.region_id
    INTO   country_check.country_id,
           country_check.country_name,
           country_check.region_id
    FROM   countries c
    WHERE  c.region_id = 2
    AND    c.country_id = country_check.country_id
    AND    ROWNUM <= 1;

    DBMS_OUTPUT.PUT_LINE(country_check.country_id);
END;
```

No ambiguity

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Conditional Control

■ IF statement

- Each IF clause contains one or more PL/SQL statements

■ CASE statement

- Each WHEN clause contains one or more PL/SQL statements
- Supports one or more conditions or one selector

■ CASE expression

- Each WHEN clause is a single expression
- Supports one or more conditions or one selector

IF Statement

Increase salary based on employees' commission

```
DECLARE
    n_employee_id          employees.employee_id%TYPE := 170;
    n_commission            employees.commission_pct%TYPE;
    n_allowance             NUMBER(4);
BEGIN
    SELECT NVL(commission_pct,0)
    INTO n_commission
    FROM employees
    WHERE employee_id = n_employee_id;

    IF TRUNC(n_commission * 100) = 0 THEN
        n_allowance := 500;
    ELSIF TRUNC(n_commission * 100) = 10 THEN
        n_allowance := 400;
    ELSIF TRUNC(n_commission * 100) = 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
    END IF;

    UPDATE employees
    SET salary = salary + n_allowance
    WHERE employee_id = n_employee_id;
END;
```

CASE Statements

Using conditions

```
CASE
    WHEN TRUNC(n_commission * 100) = 0 THEN
        n_allowance := 500;
    WHEN TRUNC(n_commission * 100) = 10 THEN
        n_allowance := 400;
    WHEN TRUNC(n_commission * 100) = 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
END CASE;
```

Using a selector

- More readable
- Only suitable for equality conditions

```
CASE TRUNC(n_commission * 100)
    WHEN 0 THEN
        n_allowance := 500;
    WHEN 10 THEN
        n_allowance := 400;
    WHEN 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
END CASE;
```

CASE Expressions

Using conditions

```
n_allowance :=  
    CASE  
        WHEN TRUNC(n_commission * 100) = 0 THEN 500  
        WHEN TRUNC(n_commission * 100) = 10 THEN 400  
        WHEN TRUNC(n_commission * 100) = 20 THEN 300  
        ELSE 200  
    END;
```

Using a selector

- Most readable
- Again, only suitable for equality conditions

```
n_allowance :=  
    CASE TRUNC(n_commission * 100)  
        WHEN 0 THEN 500  
        WHEN 10 THEN 400  
        WHEN 20 THEN 300  
        ELSE 200  
    END;
```

Iterative Control

- Allows a sequence of statements to be performed many times
 - Implemented using a LOOP statement
- Three forms of a LOOP statement
 - LOOP
 - WHILE-LOOP
 - FOR-LOOP

LOOP Statement

Syntax:

```
LOOP  
    sequence of statements  
END LOOP;
```

Requires an EXIT statement with an optional condition

```
EXIT [ WHEN condition ];
```

Example: iterate until variable i is greater than 100

```
DECLARE  
    i NUMBER(3) := 0;  
BEGIN  
    LOOP  
        i := i + 10;  
        IF i > 100 THEN  
            EXIT;  
        END IF;  
    END LOOP;  
END;
```

```
DECLARE  
    i NUMBER(3) := 0;  
BEGIN  
    LOOP  
        i := i + 10;  
        EXIT WHEN i > 100;  
    END LOOP;  
END;
```

WHILE-LOOP Statement

Syntax:

```
WHILE condition LOOP  
    sequence of statements;  
END LOOP;
```

- Evaluates a condition prior to each iteration of the loop
 - The loop is not executed if the condition is initially FALSE or NULL
- Example: iterates until the variable `i` is greater than 100

```
DECLARE  
    i NUMBER(3) := 0;  
BEGIN  
    WHILE i <= 100 LOOP  
        i := i + 10;  
    END LOOP;  
END;
```

FOR-LOOP Statement

Syntax:

```
FOR counter IN [REVERSE] lower_bound .. upper_bound LOOP  
    -- sequence of statements;  
END LOOP;
```

- Counter is implicitly declared, and value cannot be assigned to it
 - It is of type PLS_INTEGER
 - 32-bit integer, more efficient than NUMBER, only available in PL/SQL
- lower_bound and upper_bound can be variables or constants
 - They are always in the order lower_bound .. upper_bound
 - With the REVERSE key word, the counter is initialized to the upper_bound

FOR-LOOP Example

- Insert all Cartesian products of single-digit integers into a table
 - Use a two-level nested FOR-LOOP

```
BEGIN
    FOR outer_counter IN 1 .. 9 LOOP
        FOR inner_counter IN 1 .. 9 LOOP
            INSERT INTO cartesian_product
            VALUES (outer_counter, inner_counter);
        END LOOP;
    END LOOP;
END;
```

- This example illustrates the scope rules for counter variables
 - `outer_counter` variable can be referenced in the inner loop
 - `inner_counter` variable is not available in the outer loop

Exceptions

- Exception section of a block used to process errors
 - Contains one or more exception handlers
 - OTHERS key word handles all exceptions that are not explicitly named
 - Must be the last exception handler
- Common predefined exceptions
 - NO_DATA_FOUND
 - Raised when a SELECT INTO returns no rows
 - TOO_MANY_ROWS
 - Raised when a SELECT INTO returns more than one row
 - DUP_VAL_ON_INDEX
 - Raised when an inserted or updated record violates a unique index
- To stop execution, roll back database changes, and display a message:
`RAISE_APPLICATION_ERROR(error_number, error_message);`
 - `error_number` is a negative integer between –20000 and –20999
 - `error_message` is a character string up to 2048 bytes in length

Exceptions Example

- Mr. Smith has been promoted to manager of department 150
 - SQLERRM returns Oracle's error message based on the internal error raised

```
DECLARE
    n_employee_id employees.employee_id%TYPE;
BEGIN
    SELECT e.employee_id
    INTO   n_employee_id
    FROM   employees e
    WHERE  e.last_name = 'SMITH';

    UPDATE departments
    SET manager_id = n_employee_id
    WHERE department_id = 150;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20999, 'Employee does not exist');
    WHEN TOO_MANY_ROWS THEN
        RAISE_APPLICATION_ERROR(-20999, 'Multiple employees found');
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20999, SQLERRM);
END;
```

WHEN OTHERS

Be careful with WHEN OTHERS

- Using RAISE_APPLICATION_ERROR may mask an unexpected exception
- Ignoring the exception is worse!
- Consider doing any processing and then using RAISE to re-raise original exception

Use it sparingly

- Capturing families of exceptions is hard since Oracle does not have an exception hierarchy

Do not use it as part of normal processing

Far too many reasons why this may be caught, use NO_DATA_FOUND instead

```
<<find_country>>
DECLARE
    country_id    countries.country_id%TYPE := 'XX';
    country_name  countries.country_name%TYPE;
BEGIN
    SELECT c.country_name
    INTO   find_country.country_name
    FROM   countries c
    WHERE  c.country_id = find_country.country_id;

    DBMS_OUTPUT.PUT_LINE('Country found with name ' ||
                           find_country.country_name);

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Country not found');
END;
```



HANDS-ON
EXERCISE

30 min

Exercise 13.1: Building Anonymous Blocks

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Implicit Cursors

- Oracle automatically opens a cursor to process each SQL statement
 - Refers only to the last SQL statement executed
- The most recent implicit cursor is referred to as the “SQL” cursor
 - Contains attributes that provide information about the execution of `INSERT`, `UPDATE`, `DELETE`, or `SELECT INTO` statements
- Common attributes:
 - `SQL%FOUND` returns a Boolean value
 - TRUE if an `INSERT`, `UPDATE`, or `DELETE` affected one or more rows or a `SELECT INTO` returned one or more rows
 - `SQL%NOTFOUND` is the logical opposite of `SQL%FOUND`
 - `SQL%ROWCOUNT`
 - Returns the number of rows affected by an `INSERT`, `UPDATE`, or `DELETE`, or returned by a `SELECT INTO`
- Attributes can be used in procedural statements but not in SQL statements

Explicit Cursor Loop

- An Explicit Cursor is defined as a SELECT statement
- OPEN executes the query
 - Sets up the cursor in memory but does not fetch the results
- A LOOP is used to retrieve multiple rows from a cursor
 - FETCH retrieves a row from the cursor
 - Places it into a cursor record, and advances the cursor to the next row
 - EXIT terminates the loop
 - Usually based on a cursor attribute
- CLOSE deallocates memory

```
DECLARE
    CURSOR cursor_name
    IS
        select_statement;

    cursor_record cursor_name%ROWTYPE;
BEGIN
    OPEN cursor_name;

    LOOP
        FETCH cursor_name INTO cursor_record;
        EXIT WHEN cursor_name%NOTFOUND;

        ... use the cursor record ...

    END LOOP;

    CLOSE cursor_name;
END;
```

Cursor Record and Cursor Attributes

■ Cursor record

- Column values are referenced using the dot notation

```
cursor_record_name.column_name | cursor_record_name.alias_name
```

■ Cursor attributes

- cursor_name%FOUND and cursor_name%NOTFOUND
 - Indicates whether a row was fetched
- cursor_name%ROWCOUNT
 - Maintains a count of the number of rows fetched from the cursor
- cursor_name%ISOPEN
 - Indicates whether the cursor is open

Cursor Example

- Change the employee's area code from 212 to 818

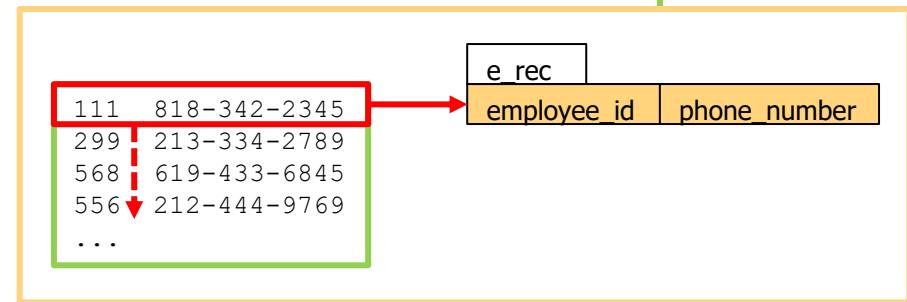
```
DECLARE
    CURSOR e_cur
    IS
        SELECT employee_id, phone_number
        FROM employees;
    e_rec e_cur%ROWTYPE;
BEGIN
    OPEN e_cur;

    LOOP
        FETCH e_cur INTO e_rec;
        EXIT WHEN e_cur%NOTFOUND;

        IF SUBSTR(e_rec.phone_number, 1, 3) = '212' THEN
            e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

            UPDATE employees
            SET phone_number = e_rec.phone_number
            WHERE employee_id = e_rec.employee_id;
        END IF;
    END LOOP;

    CLOSE e_cur;
END;
```



Cursor Parameters

- A cursor can accept input parameters
 - Used to pass information to the cursor
 - Datatype cannot have a length qualifier

```
CURSOR cursor_name(parm datatype, ..., parm datatype)
IS
    select_statement
```

- Passed to the cursor as part of the OPEN command

```
OPEN cursor_name(parameter, ..., parameter);
```

Cursor Parameters Example

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT employee_id, phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1,3) = in_area_code;

        e_rec e_cur_parm%ROWTYPE;
BEGIN
    OPEN e_cur_parm('212');

    LOOP
        FETCH e_cur_parm INTO e_rec;
        EXIT WHEN e_cur_parm%NOTFOUND;

        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE employee_id = e_rec.employee_id;
    END LOOP;

    CLOSE e_cur_parm;
END;
```

FOR-LOOP Cursors

■ Simplifies the use of cursors

- Automatically provides cursor management
 - Cursor record is implicitly declared
 - Opens the cursor when the loop is begun
 - Fetches one record for each loop iteration
 - Closes the cursor when the loop is complete

■ Syntax:

```
FOR cursor_record IN cursor_name(parameter, ..., parameter)
LOOP
    -- Cursor processing statements;
END LOOP;
```

■ Cursor record

- Used as the loop counter
- Available only inside the FOR-LOOP

FOR-LOOP Cursor Example

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT employee_id, phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1, 3) = in_area_code;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE employee_id = e_rec.employee_id;
    END LOOP;
END;
```

Using ROWID in Cursors

- Retrieve ROWID, then use it in the update instead of employee_id
 - Eliminates an index search by using ROWID to find the row

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT ROWID AS e_rowid, phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1, 3) = in_area_code;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE ROWID = e_rec.e_rowid;

    END LOOP;
END;
```

FOR UPDATE and CURRENT OF Clauses

- FOR UPDATE can be used to lock all cursor rows during the OPEN statement
 - Eliminates the possibility of waiting during the update, but may reduce concurrency

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1, 3) = in_area_code
        FOR UPDATE;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE CURRENT OF e_cur_parm;
    END LOOP;
END;
```

CURRENT OF is only permitted in FOR UPDATE cursors (no performance impact, but makes code easier to read)

Looping Over Implicit Cursors

- The FOR LOOP construct also works with implicit cursors
 - Does not allow WHERE CURRENT OF
 - Cannot refer to any cursor properties (such as %NOTFOUND)

```
BEGIN
    FOR e_rec IN (
        SELECT ROWID, phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1, 3) = '212'
    ) LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE ROWID = e_rec.ROWID;
    END LOOP;
END;
```



HANDS-ON
EXERCISE

20 min

Exercise 13.2: Using Cursors

- Please complete this exercise in your Exercise Manual

BULK COLLECT

- Although the FOR LOOP cursor is easy to use, it is not the most efficient
 - BULK COLLECT in combination with FORALL is more efficient
- Requires a PL/SQL collection
 - For example, a variable size array (variable, but the max size is fixed)
 - `TYPE emp_array IS VARRAY(batchsize) OF cur%ROWTYPE;`
 - And a variable of that type
- BULK COLLECT comes in two forms:
 - `SELECT ... BULK COLLECT INTO collection`
 - Useful if the total data volume is relatively small
 - `FETCH ... BULK COLLECT INTO collection [LIMIT batch_size]`
 - Allows the data to be processed in batches
- FORALL allows a DML command to be executed for each item in a collection

BULK COLLECT Example

```
DECLARE
    CURSOR cur(in_area_code VARCHAR2) IS
        SELECT ROWID, phone_number FROM employees WHERE SUBSTR(phone_number, 1, 3) = in_area_code;
    batchsize CONSTANT PLS_INTEGER := 1000;
    TYPE emp_array IS VARRAY(batchsize) OF cur%ROWTYPE;
    emps emp_array;
BEGIN
    OPEN cur(in_area_code => '818');
    LOOP
        FETCH cur BULK COLLECT INTO emps LIMIT batchsize;
        -- The for loop doesn't run when emps.COUNT() = 0
        FOR j IN 1..emps.COUNT() LOOP
            emps(j).phone_number := '515' || SUBSTR(emps(j).phone_number, 4);
        END LOOP;

        FORALL i IN 1..emps.COUNT()
            UPDATE employees e
            SET e.phone_number = emps(i).phone_number
            WHERE ROWID = emps(i).ROWID;

        EXIT WHEN cur%NOTFOUND;
    END LOOP;
    CLOSE cur;
END;
```

Important not to test this before processing the batch, but also recognize that the last batch may be empty

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Working with PL/SQL
- Declaring, assigning, and using scalar variables
- Defining and using conditional, iterative, and sequential control
- Utilizing error handling and writing exception handlers
- Processing result sets with cursors
- Improving cursor processing with FOR-LOOP cursors
- Improving update and delete performance by using cursors



Technology Immersion Program

Working with Relational Databases

Chapter 14: Creating Stored Procedures, Functions, and Packages

Chapter Overview

In this chapter, we will explore:

- Creating procedures and functions
- Package procedures and functions
- Debug syntax and runtime errors

Chapter Concepts

Procedures and Functions

Packages

Debugging

Chapter Summary

Named Blocks in PL/SQL

- So far, all the blocks we have seen have been *anonymous*
 - Anonymous blocks are compiled and executed when presented
- Blocks can also be *named*
 - Named blocks are subprograms (procedures and functions)
 - Executed when explicitly called from other blocks
- Subprograms can be stored in the database
 - On their own or contained inside of a package
 - Stored subprograms are visible in the session
 - And can be called by other programs

The Structure of PL/SQL Blocks

Anonymous Block

```
DECLARE
    Declaration Section
BEGIN
    Execution Section
EXCEPTION
    Exception Section
END;
```

Named Block

```
Block Header
IS | AS
Declaration Section
BEGIN
    Execution Section
EXCEPTION
    Exception Section
END;
```

Procedures and Functions

- Use the CREATE statement to store procedures and functions in the database
 - The REPLACE option recreates the program if it already exists
- Functions are the same as procedures but contain two additional clauses
 - A RETURN datatype in the definition
 - A RETURN statement in the body
 - Stops execution and returns a single value to the calling program
- Parameters are used to pass data to and from procedures and functions

parameter_name [parameter_mode] datatype [default_value_clause]

- Datatype does not include length
- Three parameter modes: IN (default mode), OUT, IN OUT
- Name parameters distinctly (p_, parm_, in_) or scope with the name of the block

- Procedures are called in a standalone statement
 - Functions are called when they are used in a statement

Procedures and Function Example

- Create a procedure to make William Smith a programmer
 - Call a function to check if the current number of IT Programmers allows him to be moved into that role

Would normally anchor these declarations, but this illustrates that parameters do not have a size

```
CREATE OR REPLACE PROCEDURE p_change_job(  
    job_id      IN VARCHAR2,  
    employee_id IN NUMBER  
)  
IS  
BEGIN  
    IF f_can_promote(p_change_job.job_id) THEN  
        UPDATE employees e  
        SET e.job_id      = p_change_job.job_id  
        WHERE e.employee_id = p_change_job.employee_id;  
    END IF;  
END;
```

```
CREATE OR REPLACE FUNCTION f_can_promote(  
    job_id IN jobs.job_id%TYPE  
) RETURN BOOLEAN  
IS  
    count_job NUMBER(3);  
BEGIN  
    SELECT COUNT(*)  
    INTO   f_can_promote.count_job  
    FROM   employees e  
    WHERE  e.job_id = f_can_promote.job_id;  
  
    RETURN (count_job < 5);  
END;
```

Calling Stored Procedures

- Call a procedure in an anonymous block, or another procedure:

```
BEGIN  
    p_change_job('IT_PROG', 171);  
END;
```

- Can also pass parameters by name:

```
BEGIN  
    p_change_job(job_id => 'IT_PROG',  
                 employee_id => 171);  
END;
```

- Especially useful if there are optional parameters:

```
BEGIN  
    p_change_job(employee_id => 171);  
END;
```

```
CREATE OR REPLACE PROCEDURE p_change_job(  
    job_id      IN jobs.job_id%TYPE := 'IT_PROG',  
    employee_id IN employees.employee_id%TYPE  
)  
IS ...
```

- But best practice in all cases

Dropping Procedures and Functions

- Use the `DROP` statement to remove a procedure or function from the database
- Syntax:

```
DROP PROCEDURE procedure_name;
```

```
DROP FUNCTION function_name;
```

- Example:
 - Remove `f_can_promote` function from the database

```
DROP FUNCTION f_can_promote;
```

Chapter Concepts

Procedures and Functions

Packages

Debugging

Chapter Summary

What Is a Package?

- Set of logically related objects
 - Groups procedures and functions
 - Stored in the database
- Example:
 - Define a package that will contain all processing associated with the employees table

Package Example

Application

```
BEGIN  
    emp_maint.delete_employee(...);  
END;
```

```
BEGIN  
    emp_maint.update_employee(...);  
END;
```

```
BEGIN  
    emp_maint.delete_employee(...);  
END;
```

Database

```
PACKAGE emp_maint IS  
  
PROCEDURE delete_employee(...)  
IS  
BEGIN  
    ...  
END delete_employee;
```

```
PROCEDURE update_employee(...)  
IS  
BEGIN  
    ...  
END update_employee;
```

```
END emp_maint;
```

Employees table

Package Components

- Each package consists of a specification and a body
 - Declared separately
- Package specification is used to declare package components
 - Declared objects are public (available to any PL/SQL block)
 - Each entry is a procedure or function declaration

```
CREATE [OR REPLACE] PACKAGE package_name  
[ IS | AS ]  
    object_declarations  
END package_name;
```

- Package body is used to define the package content
 - Contains definition and body of grouped procedures and functions
 - Must contain the code for all procedures and functions declared in the specification
 - Can also contain local procedures and functions used only within the package

```
CREATE [OR REPLACE] PACKAGE BODY package_name  
[ IS | AS ]  
    object_body_definitions  
END package_name;
```

Creating a Package: Example

- Convert the previous procedure and function into a package

Package specification contains declaration

Package body contains definitions

Items must be defined or declared before they are used

```
CREATE OR REPLACE PACKAGE pack_promote
IS
    PROCEDURE p_change_job(
        job_id IN jobs.job_id%TYPE := 'IT_PROG',
        employee_id IN employees.employee_id%TYPE
    );
END pack_promote;
```

```
CREATE OR REPLACE PACKAGE BODY pack_promote IS
    FUNCTION f_can_promote(
        job_id IN jobs.job_id%TYPE
    ) RETURN BOOLEAN
    IS
        count_job NUMBER(3);
    BEGIN
        SELECT COUNT(*)
        INTO f_can_promote.count_job
        FROM employees e
        WHERE e.job_id = f_can_promote.job_id;

        RETURN (count_job < 5);
    END f_can_promote;

    PROCEDURE p_change_job(
        job_id IN jobs.job_id%TYPE,
        employee_id IN employees.employee_id%TYPE
    )
    IS
    BEGIN
        IF f_can_promote(p_change_job.job_id) THEN
            UPDATE employees e
            SET e.job_id = p_change_job.job_id
            WHERE e.employee_id = p_change_job.employee_id;
        END IF;
    END p_change_job;
END pack_promote;
```

Calling Packaged Procedures and Functions

- A packaged procedure or function is called using the dot notation
- Syntax:

```
package_name.procedure_name(parameter1, ..., parameterN);  
  
variable_name := package_name.function_name(parameter1, ..., parameterN);
```

- Example:

```
BEGIN  
    pack_promote.p_change_job('IT_PROG', 171);  
END;
```

```
BEGIN  
    pack_promote.p_change_job(job_id => 'IT_PROG',  
                               employee_id => 171);  
END;
```

Dropping Package Specification and Body

- Use the `DROP` statement to remove package specification and/or package body
- Syntax:

```
DROP PACKAGE [BODY] package_name;
```

- Rules:
 - `BODY` is an optional key word
 - If it is specified, the package body is dropped
 - If it is not specified, both the body and specification are dropped
- Example:
 - Remove `pack_promote` body and specification

```
DROP PACKAGE pack_promote;
```

Advantages of Packages

- Modular
 - Logical group of multiple related procedures and functions
- Information hiding
 - Hides local (private) procedures and functions
- Persistent variables
 - Package variables are global variables that retain values during a session
- Better performance
 - When an object within a package is referenced, the whole package is loaded into memory; any subsequent calls to other objects within the package require no disk I/O
- Avoid dependency problems among procedures and functions
 - Can recompile the package body without invalidating procedures and functions that call it

Chapter Concepts

Procedures and Functions

Packages

Debugging

Chapter Summary

Debugging Compilation Errors

- Procedures, functions, and packages are compiled when they are created
 - A message is displayed if an error occurs when the CREATE command is issued
 - Compilation errors can be viewed using the SHOW ERRORS command

■ Syntax:

```
SHOW ERRORS [PROCEDURE | FUNCTION | PACKAGE | PACKAGE  
BODY object_name]
```

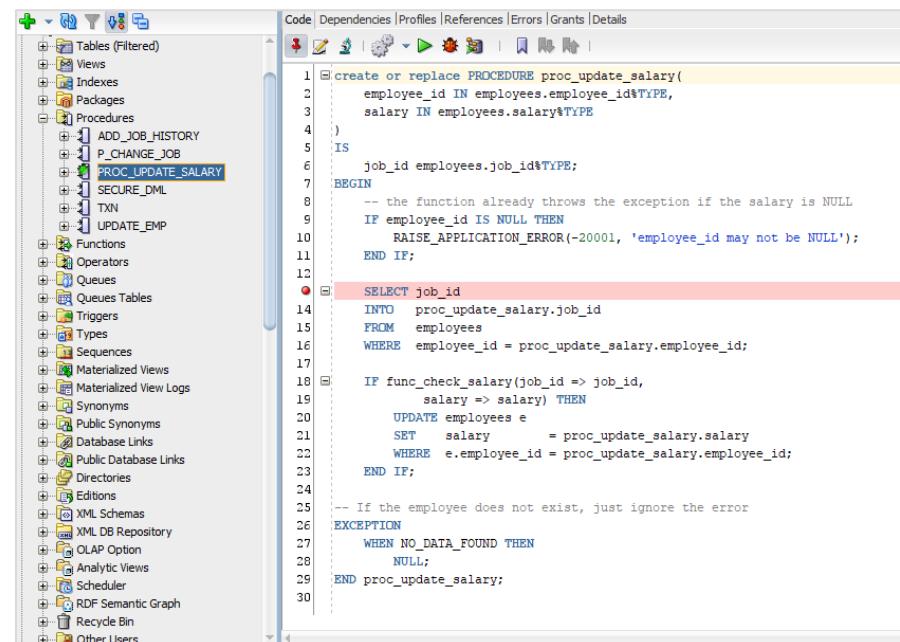
■ SHOW ERRORS command

- Shows compilation errors for the most recently created object or the named object
- Displays line and column number of the error (LINE/COL) and the error itself (ERROR)

■ SHOW ERRORS is a SQL*Plus command that queries the user_errors dictionary view

Debugging in SQL Developer

- To use the SQL Developer debugger, the procedure must be compiled in debug mode:
 - Right-click in explorer view and select **Compile for Debug**
 - Or run `ALTER PROCEDURE name COMPILE DEBUG`
- To start a debug session:
 - Right-click in explorer view and choose **DEBUG** from the menu, which will prompt for parameters
 - Write an anonymous block that sets up parameters, right-click it, and choose **DEBUG** from the menu
- Set breakpoints by opening the procedure from explorer view



The screenshot shows the SQL Developer interface. On the left, the Explorer view displays a tree of database objects, with the Procedures node expanded to show several procedures, including `PROC_UPDATE_SALARY`. On the right, the Code editor shows the PL/SQL code for the `PROC_UPDATE_SALARY` procedure. A red dot at the beginning of line 12 indicates a breakpoint has been set. The code includes logic to check if an employee ID is null and to update a salary based on a function call.

```
create or replace PROCEDURE proc_update_salary(
    employee_id IN employees.employee_id%TYPE,
    salary IN employees.salary%TYPE
)
IS
    job_id employees.job_id%TYPE;
BEGIN
    -- the function already throws the exception if the salary is NULL
    IF employee_id IS NULL THEN
        RAISE_APPLICATION_ERROR(-20001, 'employee_id may not be NULL');
    END IF;
    SELECT job_id
    INTO proc_update_salary.job_id
    FROM employees
    WHERE employee_id = proc_update_salary.employee_id;
    IF func_check_salary(job_id => job_id,
        salary => salary) THEN
        UPDATE employees e
        SET salary      = proc_update_salary.salary
        WHERE e.employee_id = proc_update_salary.employee_id;
    END IF;
    -- If the employee does not exist, just ignore the error
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        NULL;
END proc_update_salary;
```

Exercise 14.1: Stored Procedures, Functions, and Packages



30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Procedures and Functions

Packages

Debugging

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Creating procedures and functions
- Packaging procedures and functions
- Debugging syntax and runtime errors

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 15: Testing PL/SQL

Chapter Overview

In this chapter, we will explore:

- Testing in PL/SQL
 - How to approach it
- utPLSQL is a popular PL/SQL testing tool
- How can we test for updated data

Chapter Concepts

utPLSQL

Testing Updates

Chapter Summary

Why Testing in PL/SQL?

- Like in any other programming language, unit testing is essential
- Several tools available, commercial and non-commercial
- Concepts are always the same even without testing frameworks
- Three basis steps, which can be wrapped in a Testing-Package
 - StartUp: Function for building up test scenario
 - E.g., temporary tables, with or without data
 - Inserts, Updates, and Deletes for existing tables
 - Test: Function to test the Stored Procedure or Function of interest
 - TearDown: Function to cleanup
 - Rollbacks
 - Drop of tables used for testing only

utPLSQL

- One of the most popular tools is utPLSQL (<https://www.utplsql.org/>)
 - Brings JUnit or Jasmine style testing to PL/SQL
 - At least one active contributor from Fidelity in Ireland (<http://www.oraclethoughts.com/>)
- Create test package
 - Annotated with --%suite (<description>)
 - Description is optional, plain text with no quotes
 - Contains public (i.e., defined in package specification) procedures for each test
 - Annotated with --%test (<description>)
 - Other annotations to manage test fixtures and handle rollback
 - --%disabled to disable a single test spec or whole package
 - Expectations using `ut.expect()` and matchers such as `to_equal()`

utPLSQL Test Package

- Test suite annotated with
--%suite
- Specification annotated with
--%test
- Expectation uses
`ut.expect(actual)`
- Matcher is
`to_equal(expected)`
`also`
`to_(equal(expected))`

```
CREATE OR REPLACE PACKAGE test_department_manager
IS
    --%suite(Tests for func_get_department_manager)

    --%test>Returns manager for a department)
    PROCEDURE manager_for_department;

END test_department_manager;
/

CREATE OR REPLACE PACKAGE BODY test_department_manager
IS

    PROCEDURE manager_for_department
    IS
        BEGIN
            ut.expect(func_get_department_manager(10)).to_equal(200);
        END manager_for_department;

END test_department_manager;
/
```

Expectation for Exceptions

■ Annotate test with

--%throws

- With a list of exceptions by number or name

■ No expectations in the test body

```
CREATE OR REPLACE PACKAGE test_department_manager
IS
    --%suite(Tests for func_get_department_manager)
    --%test(Throws exception if department id is null)
    --%throws (-20001)
    PROCEDURE department_is_null;

END test_department_manager;
/

CREATE OR REPLACE PACKAGE BODY test_department_manager
IS

    PROCEDURE department_is_null
    IS
        result departments.manager_id%TYPE;
    BEGIN
        result := func_get_department_manager(null);
    END department_is_null;

END test_department_manager;
/
```

Running Tests

Run tests using `ut.run()`

- Needs server output to be enabled

```
SET SERVEROUTPUT ON  
BEGIN  
    ut.run();  
END;  
/  
  
BEGIN  
    ut.run('test_department_manager');  
END;  
/  
  
BEGIN  
    ut.run('test_department_manager.  
        manager_for_department');  
END;  
/
```

All tests in schema

All tests in package

One test by name

Tests for `func_get_department_manager`
Returns manager for a department [.089 sec]
Returns null when there is no manager [.008 sec]
Returns 0 when the department does not exist [.004 sec]
Throws exception if department id is null [.008 sec]

Finished in .134 seconds
4 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)
PL/SQL procedure successfully completed.

Tests for `func_get_department_manager`
Returns manager for a department [.053 sec]

Finished in .064 seconds
1 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)
PL/SQL procedure successfully completed.



30 min

Exercise 15.1: Writing PL/SQL Tests with utPLSQL

- Please complete this exercise in your Exercise Manual

Chapter Concepts

utPLSQL

Testing Updates

Chapter Summary

Test Fixtures

Provides the usual test fixture options:

- --%beforeall, --%afterall
- --%beforeeach, --%aftereach
- Used to annotate the procedure that creates or removes fixtures
- Used to annotate the package
 - Take the fixture procedure name as a parameter
 - Useful if the procedure is in another package

In addition, it includes test fixtures for a particular test

- --%beforetest, --%aftertest
- Used at the procedure level
- Take the fixture procedure name as a parameter

Grouping Test Packages

Test packages can be grouped into hierarchies

- Using the `--%suitepath(<path>)` annotation at package level
 - The `<path>` parameter is similar to a Java package, e.g., `employees.jobs`
 - A suite with the path above is a child of a suite with a path `employees`
 - Suites with the same path are treated as peers, but are separate
- Suites inherit test fixtures of their parents in the suitepath
- All tests in a suitepath can be executed together:
 - `ut.run([<schema>]:<suitepath>)`
 - Note, the schema is optional, but the colon is not

Test contexts are suites within suites

- A mechanism for grouping individual test specifications within a package
- Indicated by the `--%context(<description>)`, `--%endcontext` annotations
- Inherit the test fixtures of the suite
 - Can also have own test fixtures, executed after those of the suite
- Cannot be nested

Testing Updates

- As we have already seen, there are three ways to test updates in a database:
 - Test the database in a known state
 - Reverse all database changes
 - Run all tests in a transaction
- utPLSQL provides opportunities in all three cases
 - Test fixtures allow the database to be set into a known state
 - Fixtures for an individual test allow database changes to be reversed
 - All tests are run in a savepoint transaction, though this can be controlled
 - The `--%rollback` annotation at procedure or package level
 - Accepts a parameter `auto` (default) or `manual`

Transactions, Suites, and Fixtures

- By default (---%rollback annotation set to auto) utPLSQL creates SAVEPOINTS:
 - Before each suite
 - Before each test
- Test fixtures run after the appropriate SAVEPOINT
- If tested code performs COMMIT, need to undo changes manually

```
SAVEPOINT before_suite
beforeall

SAVEPOINT before_test
beforeeach
test
aftereach

ROLLBACK TO before_test

SAVEPOINT before_test
beforeeach
beforetest
test
aftertest
aftereach

ROLLBACK TO before_test

afterall

ROLLBACK TO before_suite
```

Asserting on Data Changes

- To properly test updates, we need to be able to check how table contents have changed
- We may expect the process to look like this:
 - Prepare expected results
 - Execute update
 - Compare expected results to actual table contents
- Until now we have seen how to compare single values, now we need to compare whole datasets

REF CURSOR

- A REF CURSOR is a reference to a cursor
 - Allows a whole result set to be passed as a parameter
 - Or returned from a stored procedure
- Unlike the cursors we have seen so far, the variable declaration does not define the query
 - Strong REF CURSORS include a definition of the return type
 - Weak REF CURSORS do not
 - Strong REF CURSORS are more type safe, but it is a small advantage
 - SYS_REF_CURSOR is a pre-defined, general purpose weak REF CURSOR
 - Because weak REF CURSORS are so much more popular than strong ones
- The creating code OPENS the cursor FOR a query
- The cursor is already open when it is received
 - Receiver of the cursor is responsible for closing it
- Cursor can only be read once

REF CURSOR Example

```
CREATE OR REPLACE PACKAGE BODY pack_employee IS
    TYPE emp_strong_t IS REF CURSOR RETURN employees%ROWTYPE;
    TYPE emp_weak_t IS REF CURSOR;

    FUNCTION count_emp(
        like_value IN VARCHAR2,
        emps_cur IN SYS_REFCURSOR
    ) RETURN PLS_INTEGER
    IS
        found PLS_INTEGER := 0;
        emp employees%ROWTYPE;
    BEGIN
        LOOP
            FETCH emps_cur INTO emp;
            EXIT WHEN emps_cur%NOTFOUND;
            IF emp.last_name LIKE like_value THEN
                found := found + 1;
            END IF;
        END LOOP;
        CLOSE emps_cur;
        RETURN found;
    END count_emp;
```

For our purposes, these are all equivalent definitions. We chose to use `SYS_REFCURSOR`, but could have used any of the others.

Already open, so cannot use a FOR LOOP

Code that receives the REF CURSOR is responsible for closing it

```
PROCEDURE prep_emps
IS
    emps_cur SYS_REFCURSOR;
    emps_count PLS_INTEGER := 0;
BEGIN
    OPEN emps_cur FOR
        SELECT *
        FROM employees;

    emps_count := count_emp('S%', emps_cur);
    DBMS_OUTPUT.PUT_LINE('Found ' || emps_count || ' records');

    END prep_emps;
END pack_employee;
```

Open the REF CURSOR, declaring the query

Pass REF CURSOR into function

Matchers to Work with REF CURSOR

■ Several utPLSQL matchers work with REF CURSOR:

- have_count (to_have_count, not_to_have_count)
- be_empty (to_be_empty, not_to_be_empty)
- equal (to_equal, not_to_equal)
 - Strict equality (data types, order of columns, order of rows)
- contain (to_contain, not_to_contain)
 - Strict contains (if the subset contains duplicated data, so must the superset)

■ The equal and contain matchers support advanced comparison options:

- include, which columns to include in the comparison
- exclude, which columns to exclude from the comparison
- unordered, ignore the order of rows, default with contains
- join_by, esp. when ignoring order, join rows by these columns to do a better diff
- unordered_columns, uc, use column names rather than column sequence

Comparing Data Before and After

```
PROCEDURE update_one_employee
IS
    test_emp_id      CONSTANT
        employees.employee_id%TYPE := 110;
    test_last_name   CONSTANT
        employees.last_name%TYPE := 'Zhang';
    unchanged_before SYS_REFCURSOR;
    unchanged_after  SYS_REFCURSOR;
    expected         SYS_REFCURSOR;
    actual           SYS_REFCURSOR;
    actual_name      employees.last_name%TYPE;
BEGIN
    -- arrange (prepare expected and before data)
    OPEN unchanged_before FOR
        SELECT * FROM employees
        WHERE employee_id != test_emp_id;

    -- It is possible to compare records, but ref
    -- cursors are easier, even with just 1 row
    OPEN expected FOR
        SELECT * FROM employees
        WHERE employee_id = test_emp_id;

    -- act (execute functionality under test)
    update_emp(employee_id => test_emp_id,
               last_name => test_last_name);
```

As an alternative, we could
create all the data here

```
-- assert (check results)
OPEN actual FOR
SELECT * FROM employees
WHERE employee_id = test_emp_id;

SELECT last_name
INTO actual_name
FROM employees
WHERE employee_id = test_emp_id;

OPEN unchanged_after FOR
SELECT * FROM employees
WHERE employee_id != test_emp_id;

ut.expect(actual)
    .to_equal(expected)
    .unordered()
    .join_by('EMPLOYEE_ID')
    .exclude('LAST_NAME');
ut.expect(actual_name)
    .to_equal(test_last_name);
ut.expect(unchanged_after)
    .to_equal(unchanged_before)
    .uc();
```

For illustration, in general
only use what is needed



HANDS-ON
EXERCISE

60 min

Exercise 15.2: Testing Updates with utPLSQL

- Please complete this exercise in your Exercise Manual

Chapter Concepts

utPLSQL

Testing Updates

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Testing in PL/SQL
 - How to approach it
- utPLSQL is a popular PL/SQL testing tool
- How can we test for updated data



Technology Immersion Program

Working with Relational Databases

Chapter 16: Creating Triggers

Chapter Overview

In this chapter, we will explore:

- Creating and using database triggers
- Controlling trigger processing using conditional predicates
- Choosing appropriate triggers to address specific requirements

Chapter Concepts

Statement-Level and Row-Level Triggers

Conditional Predicates

Managing Triggers

Chapter Summary

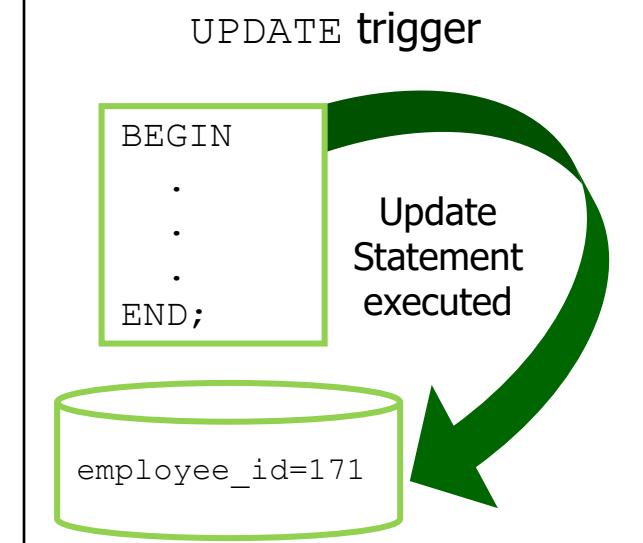
Triggers

- Defined to handle specific database events
- Defined in the database for a specific table
 - Executed on **INSERT, UPDATE, and DELETE** statements
- Contain PL/SQL code
 - Can call database stored procedures and functions
 - Can reference other database objects
- Advantages of triggers
 - Business rules are enforced in the server
 - Centralized processing
 - Do not need to enforce the same business rule in multiple front-end applications

SQL Statement

```
UPDATE employees  
SET job_id = 'IT_PROG'  
WHERE employee_id = 171;
```

Database



Trigger Types

Trigger levels

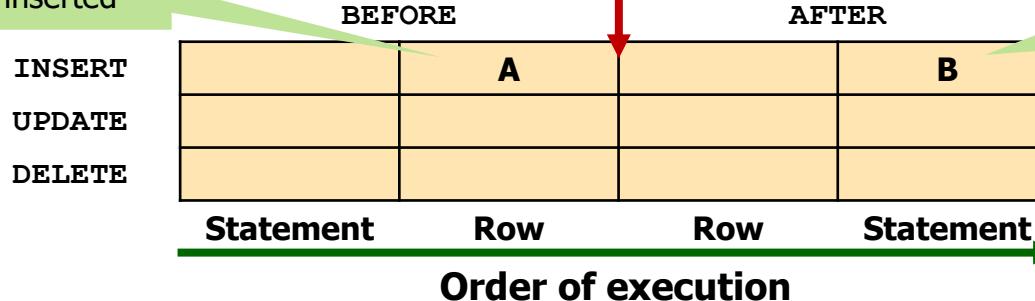
- Statement-level
 - Fires once for the SQL statement
 - Cannot reference row data
- Row-level
 - Fires once for each row affected
 - Can reference the row data

Trigger timing

- Before trigger
 - Fired before the rows are affected
 - Used to derive and modify columns
- After trigger
 - Fired after the rows are affected
 - Used to validate data

Trigger A will fire before each row is inserted

Row locked and changed



Trigger B will fire once after whole statement is complete

Guidelines for Selecting Appropriate Triggers

- Use statement triggers when row data is not required for processing
- May seem counter-intuitive, but AFTER ROW triggers are more efficient than BEFORE ROW, since the latter read the data twice (so use AFTER ROW for validation)

Usage	Example	Trigger Type
Simple business rule that can be enforced by database constraints	Only one of a pair of columns can be populated	Do not use trigger: integrity constraints are more efficient
Complex rule that cannot be enforced by database constraints	Employee can only be promoted to manager after three years employment	AFTER ROW
Compute derived column values	Calculate employee commission based on employee salary	BEFORE ROW (cannot change values in AFTER ROW)
Audit database changes	Record all changes to employee table in a journal table	AFTER ROW
Implement advanced security requirements	Employee salary cannot be changed by the employee	BEFORE statement or AFTER ROW

Creating Triggers

■ Use the CREATE TRIGGER command

- Stores the trigger in the database; does not execute it

■ Simplified syntax:

- One or more operations
- Column list is for UPDATE only:
 - Fires when any column in the list is updated
 - Defaults to all columns if not specified
- Each trigger is for a single table
- FOR EACH ROW indicates a row-level trigger
 - Omitting this clause indicates a statement-level trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER
INSERT | DELETE | UPDATE [OF column_name, ... ]
[OR INSERT| DELETE | UPDATE [OF column_name, ... ]]
[OR INSERT| DELETE | UPDATE [OF column_name, ... ]]
ON table_name
[FOR EACH ROW]
DECLARE
    local_declarations;
BEGIN
    sequence of statements
END;
```

Example of a Statement-Level Trigger

- Create a trigger that prevents any region from being deleted outside of working hours

```
CREATE OR REPLACE TRIGGER trig_region_delete
BEFORE DELETE
ON regions
BEGIN
    IF TO_CHAR(SYSDATE, 'HH24MI') NOT BETWEEN '0900' AND '1700' THEN
        RAISE_APPLICATION_ERROR(-20100, 'Not allowed to delete a region now');
    END IF;
END;
```

An exception causes the current statement to abort regardless of whether this is a BEFORE or AFTER trigger. A trigger may not execute a direct ROLLBACK.

Referencing Column Values in Triggers

- Column values can be referenced in row-level triggers using :NEW and :OLD pseudo-records
 - :NEW is used to refer to the new value of a column
 - Available only in INSERT or UPDATE triggers
 - :OLD is used to refer to the previous value of a column
 - Available only in UPDATE or DELETE triggers
 - Pseudo-records are declared and initialized automatically
 - Pseudo-records are available in BEFORE and AFTER row-level triggers
 - Values of :NEW may be changed in BEFORE triggers
- Syntax:
 - :NEW.column_name or :OLD.column_name
- SQL Developer prompts for “bind variable” values when it encounters :NEW or :OLD
 - Add SET DEFINE OFF to your script (outside the trigger definition) to disable this behavior

Example of a Row-Level Trigger

- Create a trigger that prevents any regions from being deleted outside of working hours in the U.S. and Europe

```
CREATE OR REPLACE TRIGGER trig_region_delete_row
AFTER DELETE
ON regions
FOR EACH ROW
BEGIN
    IF (:OLD.location LIKE 'Europe%'
        AND TO_CHAR(SYSDATE, 'HH24MI') NOT BETWEEN '0100' AND '0900')
        OR (:OLD.location LIKE 'US%'
        AND TO_CHAR(SYSDATE, 'HH24MI') NOT BETWEEN '0900' AND '1700') THEN
        RAISE_APPLICATION_ERROR(-20100, 'Not allowed to delete a region now');
    END IF;
END;
```

Use AFTER row-level triggers
to enforce business rules

Trigger Example: Derivation

- Create a trigger that calculates the commission of an employee based on the salary when the salary field is updated, and the new salary is less than the old salary

```
CREATE OR REPLACE TRIGGER trig_employee_update
BEFORE UPDATE OF sal
ON emp
FOR EACH ROW
BEGIN
    IF :NEW.sal < :OLD.sal THEN
        :NEW.comm := :NEW.sal * 0.25;
    END IF;
END;
```

Derivation in BEFORE trigger

Trigger Example: Journal

- Create a trigger on the region table that logs the delete when a region is removed

```
CREATE OR REPLACE TRIGGER trig_region_del
AFTER DELETE
ON regions
FOR EACH ROW
BEGIN
    INSERT INTO region_purged (
        region_id, region_name, deleted_by, delete_date
    )
    VALUES (
        :OLD.region_id, :OLD.region_name, USER, SYSDATE
    );
END;
```

Chapter Concepts

Statement-Level and Row-Level Triggers

Conditional Predicates

Managing Triggers

Chapter Summary

Conditional Predicates

- Boolean values used to determine the type of data operation that fired the trigger
- Valid values: INSERTING, UPDATING, UPDATING(column), DELETING
- Used in a trigger that is defined for multiple data operations
- Create a trigger that prevents any employee from being deleted or updated after 5:00 p.m.

```
CREATE OR REPLACE TRIGGER trig_employee_delete
AFTER DELETE OR UPDATE
ON employees
BEGIN
    IF TO_CHAR(SYSDATE, 'HH24MI') > '1700' THEN
        IF DELETING THEN
            RAISE_APPLICATION_ERROR(-20100, 'Not allowed to delete an employee now');
        ELSIF UPDATING THEN
            RAISE_APPLICATION_ERROR(-20101, 'Not allowed to update an employee now');
        END IF;
    END IF;
END;
```

Chapter Concepts

Statement-Level and Row-Level Triggers

Conditional Predicates

Managing Triggers

Chapter Summary

Enabling and Disabling Triggers

- Once created, a trigger is enabled automatically
- Enable or disable a specific trigger using the ALTER TRIGGER statement
 - Syntax:
ALTER TRIGGER trigger_name ENABLE | DISABLE;
 - Example: disable the delete trigger on the employees table
ALTER TRIGGER trig_employee_delete DISABLE;
- Enable or disable all triggers for a table using the ALL TRIGGERS option
 - Syntax:
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
 - Example: enable all triggers on the employees table
ALTER TABLE employees ENABLE ALL TRIGGERS;

Dropping Triggers

- A trigger can be removed from the database
 - Use the `DROP TRIGGER` command

■ Syntax:

```
DROP TRIGGER trigger_name;
```

- Example:
 - Drop the delete trigger on the `employees` table

```
DROP TRIGGER trig_employee_update;
```



Exercise 16.1: Working with Triggers

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Statement-Level and Row-Level Triggers

Conditional Predicates

Managing Triggers

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Creating and using database triggers
- Controlling trigger processing using conditional predicates
- Choosing appropriate triggers to address specific requirements

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 17: Data Definition Language

Chapter Overview

In this chapter, we will explore:

- Creating and managing tables
- Altering tables
- Using sequence generators
- Managing integrity constraints
- Creating and managing views
- Working with indexes

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Schema

- “Schema” is a relational database word meaning:
 - Collection of database objects
 - Including structures such as:
 - Tables
 - Views
 - Stored procedures
 - Indexes
- A schema has the name of the user who owns it
 - The terms schema, user, and account tend to be used interchangeably
 - HR and SCOTT are the users/schemas we have been using

Schema Objects

- Everything in the database *must* be owned by a user
 - The collection of objects owned by a user is called the schema
- A schema object is owned by the user account it is created in
 - Qualified by: `user_name.table_name`
 - Example: `hr.employees`
- Legal schema object names must conform to the Oracle standard
 - Maximum of 30 positions
 - Must start with an alpha character
 - Can include numbers and the special characters: \$ _ #

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

CREATE TABLE

- The simplified syntax is:

```
CREATE TABLE table_name (
    column_name datatype [DEFAULT default_value] [NOT NULL]
    [, ... ]
)
```

- A table must have at least one column
 - Each column must have a valid datatype
 - Can be defined as being mandatory
 - Can have a DEFAULT value set
- CREATE TABLE is a DDL command
 - DDL commands automatically commit
 - Any DDL command automatically commits any outstanding transaction

CREATE TABLE Example

- A table to store client information

```
CREATE TABLE clients (
    client_id          NUMBER(5) NOT NULL
, client_name        VARCHAR2(20)
, client_type        NUMBER(2) DEFAULT 01
, client_start_date DATE
);
```

Table created.

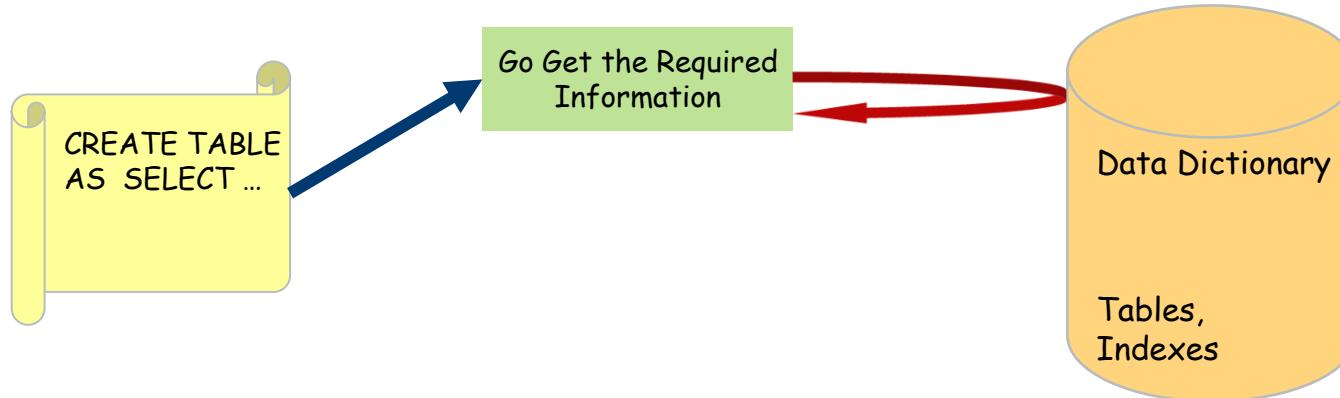
- NOT NULL is an integrity constraint specifying that the column is mandatory
- The client type is set to 01, if a value is not provided when a row is inserted
 - Using the key word DEFAULT in an UPDATE or INSERT will cause this value to be used

CREATE TABLE AS SELECT: CTAS

- IF you want to base the definition of a new table upon an existing one, the CTAS statement can be used
- Syntax:

```
CREATE TABLE table_name AS SELECT ...
```

- The necessary column specifications are retrieved from the data dictionary



CTAS: Example 1

- An image copy of an existing table can be made
 - Notice that the rows of data are brought over

```
CREATE TABLE new_jobs AS
    SELECT *
        FROM jobs;

Table created.

SELECT *
FROM new_jobs;

JOB_ID      JOB_TITLE                      MIN_SALARY MAX_SALARY
-----      -----
AD_PRES     President                     20000     40000
AD_VP       Administration Vice President 15000     30000
AD_ASST     Administration Assistant     3000      6000
FI_MGR      Finance Manager                8200      16000
...
19 rows selected.
```

CTAS: Example 1 (Continued)

- IF you wanted a copy of the definition of the table without any data, write the WHERE clause to exclude all data

```
CREATE TABLE new_jobs AS  
SELECT *  
FROM jobs  
WHERE 1 = 2;
```

Table created.

```
SELECT * FROM new_jobs;
```

No rows selected.

CTAS: Example 2

- We can also specify which columns we want and their names by providing a column list

```
CREATE TABLE new_jobs (title, job_number) AS  
    SELECT job_title, job_id  
    FROM jobs;
```

Table created.

```
SELECT * FROM new_jobs;
```

TITLE	JOB_NUMBER
President	AD_PRES
Administration Vice President	AD_VP
Administration Assistant	AD_ASST
Public Relations Representative	PR REP
...	

19 rows selected.

CTAS: Example 3

- We can also limit the rows in the new table by specifying a WHERE clause

```
CREATE TABLE new_jobs (title, job_number) AS  
    SELECT job_title, job_id  
        FROM jobs  
       WHERE job_id LIKE 'IT%';
```

Table created.

```
SELECT * FROM new_jobs;
```

TITLE	JOB_NUMBER
Programmer	IT_PROG

CTAS: Example 4

- The table being created does not have to be based upon a single table
 - Any SELECT statement can be used

```
CREATE TABLE jobs_employees AS
    SELECT j.job_title, e.last_name
    FROM jobs j
    JOIN employees e
    USING (job_id)
    WHERE job_id LIKE 'IT%';
```

Table created.

```
SELECT * FROM jobs_employees;
```

JOB_TITLE	LAST_NAME
Programmer	Hunold
Programmer	Ernst
Programmer	Austin
Programmer	Pataballa
Programmer	Lorentz

Other Table Commands

Tables can be renamed

```
RENAME table_name TO new_table_name  
RENAME jobs TO regional_jobs;
```

Be careful:

- Permissions and references to the original table are not automatically maintained

Tables can have their rows quickly removed

```
TRUNCATE TABLE new_jobs;
```

- Accomplishes the same result as `DELETE` without the `WHERE` clause

Be careful:

- `TRUNCATE` is a DDL command and cannot be rolled back
- The reason why it is so efficient

Dropping Tables

- To remove the definition of the table (not just the rows):

```
DROP TABLE table_name  
DROP TABLE new_jobs;
```

- This removes the table and all dependencies
 - Indexes, permissions, etc.
 - Code references become invalid

Restoring Dropped Tables

- Beginning in Oracle10g, the table is not actually dropped
 - The table is renamed and “held” in the recycle bin
 - Along with dependent objects, indexes, triggers, constraints
- The table can be restored using a FLASHBACK command
 - FLASHBACK TABLE table_name TO BEFORE DROP
 - FLASHBACK new_jobs TO BEFORE DROP;
 - Dependent objects, except for foreign keys, are also restored
- The table can also be renamed as a part of the flashback operation
 - FLASHBACK new_jobs TO BEFORE DROP RENAME TO some_jobs;
- Unless purged, the contents of the recycle bin will be available as long as space exists in the area where the original table was stored
- The recycle bin can be bypassed during the DROP by appending the PURGE option

```
DROP TABLE new_jobs PURGE;
```

Working with the Recycle Bin

To examine the recycle bin

```
SELECT ORIGINAL_NAME, OBJECT_NAME, TYPE, DROPTIME, CAN_UNDROP FROM user_recyclebin;
```

ORIGINAL_NAME	OBJECT_NAME	TYPE	DROPTIME	CAN_UNDROP
NEW_JOBS	BIN\$Y/ym7LmtSh2IYYBGAoBqoQ==\$0	TABLE	2018-05-03:09:36:53	YES

- Can access objects directly from the bin

```
SELECT * FROM "BIN$Y/ym7LmtSh2IYYBGAoBqoQ==$0";
```

A specific table can be removed from the recycle bin

- Or, if there was more than one matching table:

```
PURGE TABLE new_jobs;
```

```
PURGE TABLE "BIN$Y/ym7LmtSh2IYYBGAoBqoQ==$0";
```

The entire contents of the recycle bin can be purged with a single command

- Verify the contents first
- Requires DBA privileges

```
PURGE DBA_RECYCLEBIN;
```

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Altering Tables

- A table, once created, can have many of its settings changed via the ALTER command
- For example, to add a new column to an existing table:

```
ALTER TABLE table_name  
ADD (column_name datatype [DEFAULT default_value] [NOT NULL]  
[, ... ] )
```

```
ALTER TABLE new_jobs  
ADD (effective_date DATE);  
Table altered.
```

```
DESCRIBE new_jobs
```

Name	Null?	Type
JOB_ID		VARCHAR2 (10)
JOB_TITLE	NOT NULL	VARCHAR2 (35)
MIN_SALARY		NUMBER (6)
MAX_SALARY		NUMBER (6)
EFFECTIVE_DATE		DATE

Can only add a NOT NULL column if the table is empty, or there is a default specified

Altering Tables: Modifying Columns

- Using a similar syntax, existing columns can be modified
- Example:

```
ALTER TABLE new_jobs  
MODIFY (max_salary NUMBER(8));
```

Table altered.

```
DESCRIBE new_jobs
```

Name	Null?	Type
JOB_ID		VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(8)
EFFECTIVE_DATE		DATE

Restrictions on Modifying Columns

- If rows already exist, then the pre-existing data must already agree with the modification
- Examples:
 - A column's size cannot be made smaller than the existing data
 - A column's datatype cannot be changed as long as there is existing data in the column
 - Any existing constraints must be adhered to

```
ALTER TABLE new_jobs
MODIFY (job_title VARCHAR2(4));
(job_title VARCHAR2(4) )
*
ERROR at line 2:
ORA-01441: cannot decrease column length because some value is too big

ALTER TABLE new_jobs
MODIFY (job_id NUMBER(6));
(job_id NUMBER(6) )
*
ERROR at line 2:
ORA-01439: column to be modified must be empty to change datatype
```

Altering Tables: Removing Columns

- Columns can be removed from the table in one of several ways
 - Remove the definition and the data

```
ALTER TABLE new_jobs
DROP (min_salary, max_salary) CASCADE CONSTRAINTS;
```

Table altered.

```
DESCRIBE new_jobs
Name          Null?    Type
-----        -----
JOB_ID        VARCHAR2(10)
JOB_TITLE     NOT NULL VARCHAR2(35)
EFFECTIVE_DATE           DATE
```

- The CASCADE clause is needed when the column is a Foreign Key
 - In this example, the clause is unnecessary

Altering Tables: Setting Columns Unused

- The data dictionary can be modified without physically removing the column's data
 - The SET UNUSED clause

```
ALTER TABLE new_jobs  
SET UNUSED (effective_date) CASCADE CONSTRAINTS;
```

Table altered.

```
DESCRIBE new_jobs
```

Name	Null?	Type
JOB_ID		VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)

- Allows the data to be physically removed when convenient

```
ALTER TABLE new_jobs  
DROP UNUSED COLUMNS;  
Table altered.
```

Altering Tables: Renaming Columns

- Existing columns can be renamed with the RENAME clause

```
ALTER TABLE new_jobs  
RENAME COLUMN job_title TO title;
```

```
Table altered.
```

```
DESCRIBE new_jobs
```

Name	Null?	Type
JOB_ID		VARCHAR2 (10)
TITLE	NOT NULL	VARCHAR2 (35)

- Any dependent constraints are maintained
- Dependent code (triggers, views, stored procedures, etc.) are invalidated

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Sequence Generator

- A sequence generator is a database object
 - It is an Oracle-managed series of numbers
 - Often used to provide the values for primary keys
 - Remember, primary keys must be unique
- Shortened (simplified) syntax:

```
CREATE SEQUENCE sequence_name  
INCREMENT BY interval  
START WITH start_value
```

```
CREATE SEQUENCE seq_emp  
INCREMENT BY 1  
START WITH 8000;
```

Sequence created.

Sequence Generator Example

- We can now use this sequence to provide the data value for the primary key (`empno`) of the `emp` table
- Get the next sequence value by using `NEXTVAL`

```
INSERT INTO emp (empno, ename)
VALUES (seq_emp.NEXTVAL, 'CHAVAS');
```

1 row created.

- Now retrieve the rows

```
SELECT empno, ename FROM emp WHERE empno > 7900;
EMPNO ENAME
-----
7902 FORD
7934 MILLER
8000 CHAVAS
```

- A sequence number, once consumed, cannot be put back (even if rollback occurs)

IDENTITY Columns

- Sequences are normally combined with triggers to populate key fields automatically
 - Starting with Oracle 12c, IDENTITY columns make this functionality more accessible

```
GENERATED [ ALWAYS | BY DEFAULT [ ON NULL ] ] AS IDENTITY [ ( options ) ]
```

- ALWAYS means it will always be auto-generated, specifying a value is an error
- BY DEFAULT means it will be auto-generated unless specified
- BY DEFAULT ON NULL means auto-generated if set to NULL
- options are the same as for a SEQUENCE

```
CREATE TABLE clients (
    client_id          NUMBER GENERATED ALWAYS AS IDENTITY
    , client_name       VARCHAR2(20)
    , client_type       NUMBER(2) DEFAULT 01
    , client_start_date DATE
);
```

Table created.

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Integrity Constraints

- Declarative method of enforcing business rules in the database
- Defined with the corresponding table using the CREATE TABLE or ALTER TABLE command
- Validated when constraints are defined, and on each INSERT, UPDATE, and DELETE
- Five types
 - NOT NULL
 - Primary key
 - Unique key
 - Foreign key
 - Check

NOT NULL Constraints

- Enforces that a column contains a value for each row
- Usually defined in the CREATE TABLE command, as discussed in the previous section
- Example:
 - empno is mandatory

```
CREATE TABLE emp (
    empno NUMBER(4)      NOT NULL
    , ename VARCHAR2(10)
    ,
    );
    ...
```

- May be used with ALTER TABLE MODIFY after adding a new column to a populated table
 - Add column without the constraint
 - Populate the column
 - Alter the table to add the constraint

Primary Key Constraints

- Ensures that no two rows of a table have duplicate values in the key column(s)
 - All columns in the primary key must be mandatory
 - Only one primary key is allowed per table
- For ease of maintenance, use the `ALTER TABLE` command with the following syntax:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
PRIMARY KEY (column_name, ..., column_name);
```

Primary Key Constraints (continued)

- An ideal primary key column is one whose value rarely changes
 - Updating a primary key usually impacts other tables
- A common naming convention is to end primary keys with the `_pk` suffix
 - Makes it easier to identify these keys when we query the data dictionary
- Example:
 - `empno` is used to uniquely identify employees

```
ALTER TABLE emp
ADD CONSTRAINT emp_pk
PRIMARY KEY (empno);
```

- Short numeric columns are preferred as primary keys because they offer the best performance and require the least amount of storage

Unique Key Constraints

- Same rules as for primary key, but allow optional columns
 - Allow an unlimited number per table
 - A common standard is to end unique keys with the `_uk` suffix
- For ease of maintenance, use the `ALTER TABLE` command with the following syntax:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
UNIQUE (column_name, ..., column_name);
```

Example:

- `ename`, `job`, and `hiredate` must be unique

```
ALTER TABLE emp
ADD CONSTRAINT emp_uk1
UNIQUE (ename, job, hiredate);
```

Foreign Key Constraints

- Ensures that every value of a foreign key in the child table exists in the parent table
 - Prevents updates and deletes of parent table that would violate the constraint

```
ALTER TABLE child_table_name  
ADD CONSTRAINT constraint_name  
FOREIGN KEY (column_name, ..., column_name)  
REFERENCES parent_table_name [(column_name, ..., column_name)]  
[ ON DELETE CASCADE | ON DELETE SET NULL ];
```

- A foreign key can only reference existing primary or unique keys of the parent table
 - If no parent columns are specified, references primary key
- The ON DELETE options:
 - CASCADE automatically deletes child records on deletion of parent record
 - SET NULL automatically updates child columns to NULL on deletion of parent record
- A common naming convention is to end foreign keys with the `_fk` suffix

Foreign Key Constraint Examples

- Each employee must belong to a department that is defined in the dept table
 - A department cannot be deleted if employees work there

```
ALTER TABLE emp
ADD CONSTRAINT emp_dept_fk1
FOREIGN KEY (deptno) REFERENCES dept (deptno);
```

- Each employee must belong to a department that is defined in the dept table
 - If a department is being deleted, all of its employees should be deleted

```
ALTER TABLE emp
ADD CONSTRAINT emp_dept_fk1
FOREIGN KEY (deptno) REFERENCES dept (deptno)
ON DELETE CASCADE;
```



Make sure this
is really what
you want!

Check Constraints

- Enforces business rules on update or insert
 - Limited to simple validation based on constants or column values in the row

```
ALTER TABLE table_name  
ADD CONSTRAINT constraint_name  
CHECK (condition);
```

- A common naming convention is to end check constraints with the `_ck` suffix
- E.g., Commission must always be less than 10% of salary

```
ALTER TABLE emp  
ADD CONSTRAINT emp_comm_sal_ck  
CHECK (comm < sal * .10);
```

Dropping Constraints

- Drop constraints using the following syntax:

```
ALTER TABLE table_name  
DROP CONSTRAINT constraint_name;
```

- For example, drop emp_comm_sal_ck check constraint

```
ALTER TABLE emp  
DROP CONSTRAINT emp_comm_sal_ck;
```

- A simpler way to drop primary key constraints is using this syntax:

```
ALTER TABLE table_name  
DROP PRIMARY KEY [CASCADE];
```

- CASCADE option drops all the related foreign key constraints
- For example, drop the primary key on emp table

```
ALTER TABLE emp  
DROP PRIMARY KEY;
```

Enabling and Disabling Constraints

- Constraints can be made temporarily inactive and can later be reactivated using the following syntax:

```
ALTER TABLE table_name  
  DISABLE CONSTRAINT constraint_name;
```

```
ALTER TABLE table_name  
  ENABLE CONSTRAINT constraint_name;
```

■ Examples:

- Disable `emp_dept_fk1` foreign key constraint, then re-enable it

```
ALTER TABLE emp DISABLE CONSTRAINT emp_dept_fk1;
```

Table EMP altered

```
ALTER TABLE emp ENABLE CONSTRAINT emp_dept_fk1;
```

Table EMP altered

Constraints in the Dictionary

- Constraints are shown in the `user_constraints` dictionary view
 - Constraint type is "P" for primary key, "U" for unique key, "R" for referential constraint or foreign key, "C" for check constraint, and "V" for view with check option
 - Constraint status is either `ENABLED` or `DISABLED`
 - Constraint columns are stored in `user_cons_columns` view

■ Example:

```
SELECT constraint_name, constraint_type, status
FROM user_constraints
WHERE table_name = 'EMP';
```

CONSTRAINT_NAME	C STATUS
EMP_PK	P ENABLED
EMP_DEPTNO_FK1	R ENABLED

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Views

- A view is a stored definition of data
 - The data dictionary is updated when a view is created
 - The data is not physically stored in a view
 - It is read from the tables the view references
- Syntax:

```
CREATE OR REPLACE VIEW view_name (column_name, column_name ...) AS  
SELECT ...  
[WITH { READ ONLY } | { CHECK OPTION [CONSTRAINT constraint_name] } ]
```

- The SELECT statement can be any valid query
- The view can specify the column names
- There are two possible restrictions
 - READ ONLY means the view data cannot be updated
 - CHECK OPTION restricts updates to values that would be selected by the view

Reasons to Create Views

- Security
 - To restrict the columns that can be seen
 - To restrict the rows that can be read
- To simplify the accessing of information
 - Complex SQL statements do not have to be continually recreated
- To provide typical application images of the information
 - Including using a different column name
 - Preferred by different groups
- Isolate applications from table changes

VIEW Example

- Provide a restricted view of the jobs table for other accounts
 - Only three of the columns and some of the rows in the jobs table can be seen
 - So that a restricted user can only access the data in the view

```
CREATE OR REPLACE VIEW vw_jobs (max_sal, min_sal, title) AS
SELECT max_salary, min_salary, job_title
FROM jobs
WHERE max_salary < 8000
WITH CHECK OPTION CONSTRAINT ck_restrict_jobs;
```

View created.

```
SELECT * FROM vw_jobs;
    MAX_SAL      MIN_SAL TITLE
-----
      6000        3000 Administration Assistant
      5500        2500 Purchasing Clerk
      5000        2000 Stock Clerk
      5500        2500 Shipping Clerk
```

Updatable Views

- Views may be updatable if they contain a key-preserved table
 - A table where each row appears only once in the view results
- Each operation on an updatable view may only modify the data of one underlying table

```
UPDATE vw_jobs
SET max_sal = 7500
WHERE title = 'Administration Assistant';

1 row updated.

UPDATE vw_jobs
SET max_sal = 8500
WHERE title = 'Administration Assistant';

Error report -
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Updatable Views (continued)

- To find out if a view column is updatable

```
SELECT table_name, column_name, updatable  
FROM   user_updatable_columns  
WHERE  table_name = 'VW_JOBS';
```

TABLE_NAME	COLUMN_NAME	UPD
VW_JOBS	MAX_SAL	YES
VW_JOBS	MIN_SAL	YES
VW_JOBS	TITLE	YES

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

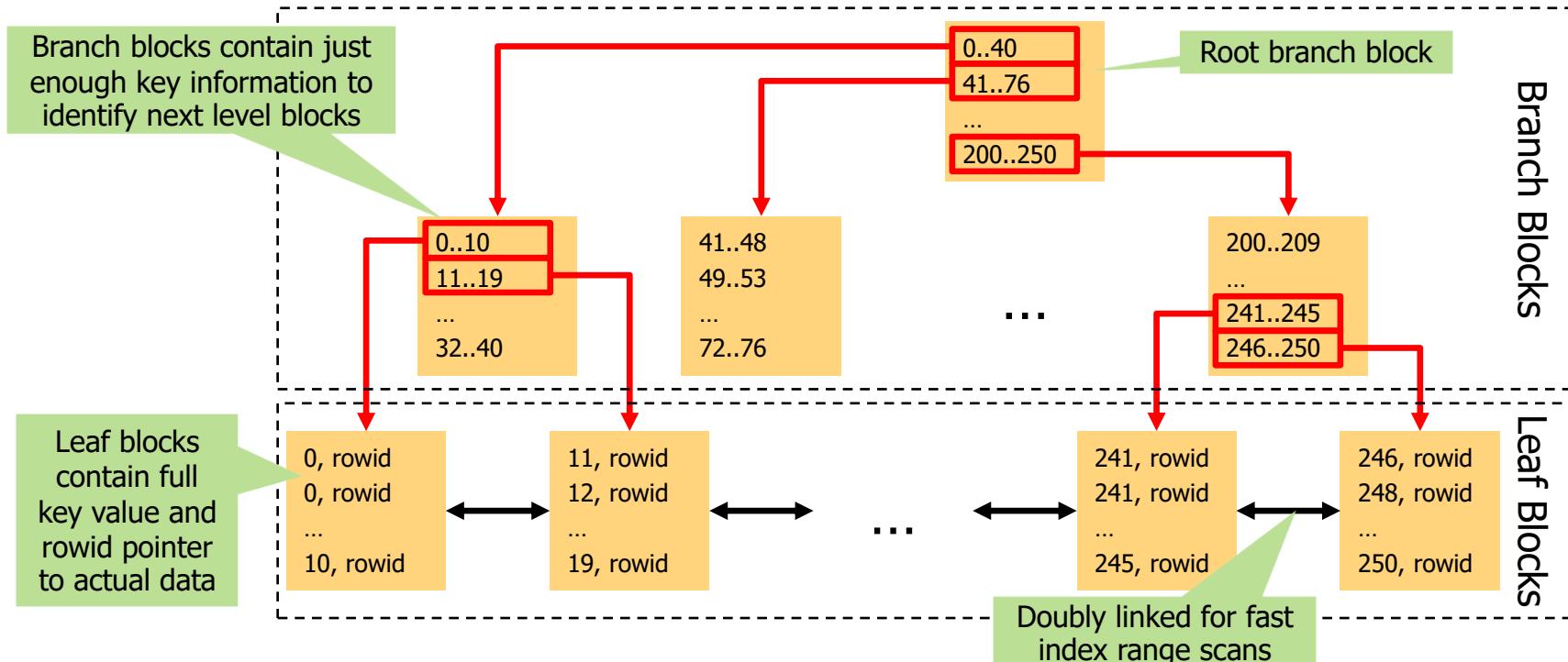
Chapter Summary

Indexes

- An index is a schema object for improving query performance
 - Stored separately from the table
 - Not referenced in SQL, used by query optimizer
- Indexes work because keys are generally much smaller than the data they reference
 - Index stores only the key and a pointer to the actual data (rowid)
 - Index can be traversed much faster than the data
 - Oracle automatically creates indexes for primary key and unique constraints
- Default Oracle index type is B-Tree index
 - “Balanced” tree because each leaf node is the same number of steps away from the root
 - Accessing leaf data requires $O(\log N)$ block accesses (increases very slowly)

B-Tree Indexes

- Imagine a numeric index, e.g., foreign key index for department_id in employees



Unique Index

- An index with a one-to-one relationship to table rows
 - Efficient because each leaf entry points to one row in the table
 - Automatically generated for each primary key and unique key constraint
 - Unless there is already a covering index
 - They are dropped when the underlying constraint is dropped or disabled

```
ALTER TABLE emp  
ADD CONSTRAINT emp_pk  
PRIMARY KEY (empno);
```

```
ALTER TABLE emp  
ADD CONSTRAINT emp_uk1  
UNIQUE (ename, hiredate);
```

- Or you can create your own

```
CREATE UNIQUE INDEX emp_pk_ix  
ON emp (empno);
```

```
CREATE UNIQUE INDEX emp_uk1_ix  
ON emp (ename, hiredate);
```

- Indexes are frequently named with a suffix `_ix`

Non-Unique Index

- A non-unique index has values that point to one or more table rows
 - More than one rowid per key value
 - They are generally less efficient than unique indexes
 - How much less efficient depends on the selectivity of the columns

```
CREATE INDEX index_name ON table_name (column_name, ..., column_name)
```

Example:

- Manager in the emp table is not unique
 - More than one employee can work for the same manager
- If we frequently asked the question “who works for...”, we might create an index to improve performance

```
CREATE INDEX emp_mgr_ix  
ON emp (mgr);
```

Dropping Indexes

- Any index may be dropped unless it is being used to enforce a constraint

```
DROP INDEX emp_mgr_ix;
```

- Automatically created indexes for primary keys and unique constraints are automatically dropped if the constraint is dropped
 - You cannot usually drop them manually
- Any index you create may be used by a constraint that is created later
 - In this case, you will not be able to drop the index without first dropping the constraint

Function-Based Indexes

- You can create an index that includes a function call
 - SQL scalar function
 - User-defined function
 - Package function
- Oracle pre-computes the function return value and stores it in the index

```
CREATE INDEX emp_fname_uppercase_ix  
ON employees (UPPER(first_name));
```

- What do you think the performance impact of this will be:
 - For queries?
 - For data maintenance (insert and update)?

Indexing Guidelines

- Deciding WHAT indexes to create is driven by HOW the data is accessed
 - This can change over time
- Columns to consider for an index
 - All columns that are part of primary and unique keys are indexed automatically
 - Foreign keys that are commonly used for joins
 - Frequently queried columns that are relatively unique (high selectivity)
 - Return less than 10 percent of the rows when queried
- Columns not to index
 - Columns that are frequently updated
 - Columns in small tables
 - Columns that contain few unique values (low selectivity)
 - Reason: likely to be retrieving many rows



HANDS-ON
EXERCISE

45 min

Exercise 17.1: Table Management

- Please turn to the Exercise Manual and complete this exercise

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

DDL Through SQL Developer

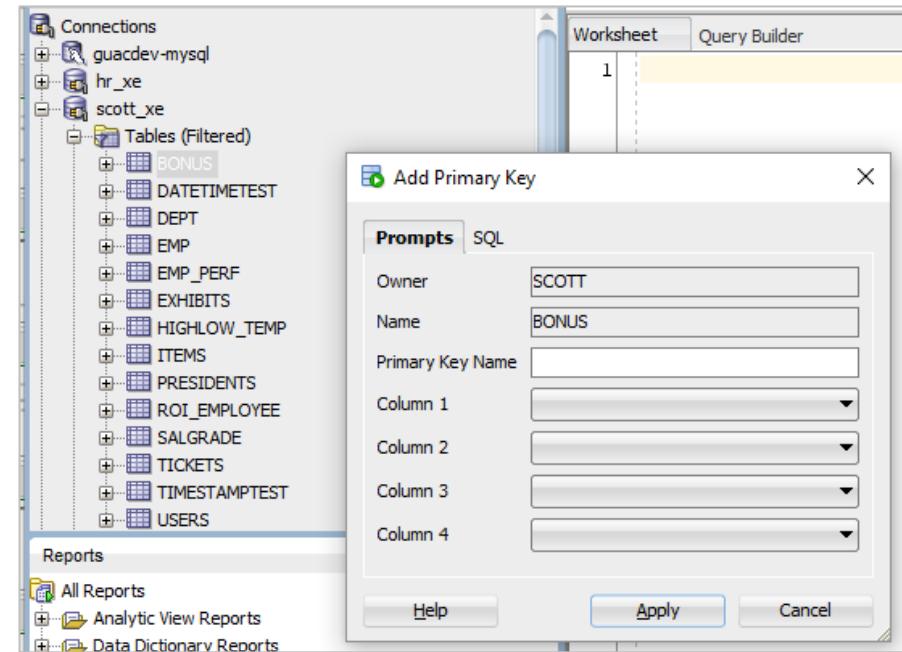
- SQL Developer provides a method of doing most DDL through the GUI

- Do not use this feature***

- Get in the habit of typing the commands:

- It helps you learn
- It means there is a clear history
- It allows you to maintain a DDL file

- If you are unsure of the syntax, use SQL Developer and then immediately export the DDL to examine it and to keep a history



Maintaining DDL Files

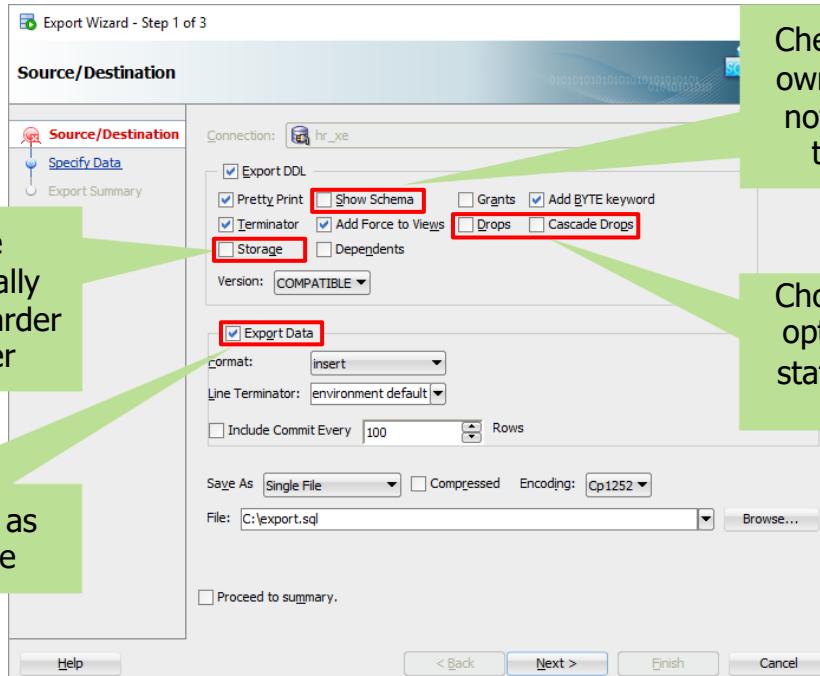
- You should maintain a series of DDL files for your project
 - You should be able to completely recreate the database from these scripts
 - Separate test data from schema structure
 - Consider separating schema objects into separate files and including them using @@

```
@@client_ddl.sql  
@@staff_ddl.sql  
  
@@populate_data.sql
```

<https://docs.oracle.com/en/database/oracle/oracle-database/12.2/sqplug/SQL-Plus-command-reference.html>

Exporting from SQL Developer

- You can generate a starting point by exporting from SQL Developer
 - Experiment with the options



Adds information about the physical storage that you usually leave to Oracle and makes it harder to apply on a different server

If you want the data as well as the structure

Checking this option shows schema ownership for each object, which is not useful if you want to re-create the objects in another schema

Choosing one of these options creates DROP statements before the CREATES

Chapter Concepts

Schemas

Create and Manage Tables

Alter a Table

Creating Sequence Generators

Enforcing Database Integrity

Managing Views

Managing Indexes

Using SQL Developer

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Creating and managing tables
- Altering tables
- Using sequence generators
- Managing integrity constraints
- Creating and managing views
- Working with indexes

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 18: Scaling Relational Databases

Chapter Overview

In this chapter, we will explore:

- The factors involved in scaling relational databases
- How relational databases can be deployed to cloud systems

Chapter Concepts

Scaling Relational Databases

Relational Databases in the Cloud

Chapter Summary

Types of Scaling

- Data Volume
 - Total amount of data stored, measured in millions of rows or terabytes (or petabytes+)
- Throughput
 - Number of data operations per second
 - Usually focuses on the number of inserts or updates
- Concurrency
 - Often driven by the number of users working on the system simultaneously
 - Related to throughput, but the emphasis is that the operations occur at the same time
- Resilience
 - Tolerate a certain number of failures (hardware, infrastructure)
- Compute
 - Millions (or billions) of floating point operations per second (megaflops or teraflops)
 - Not usually an issue for database systems

How Relational Databases Scale – Data Volume

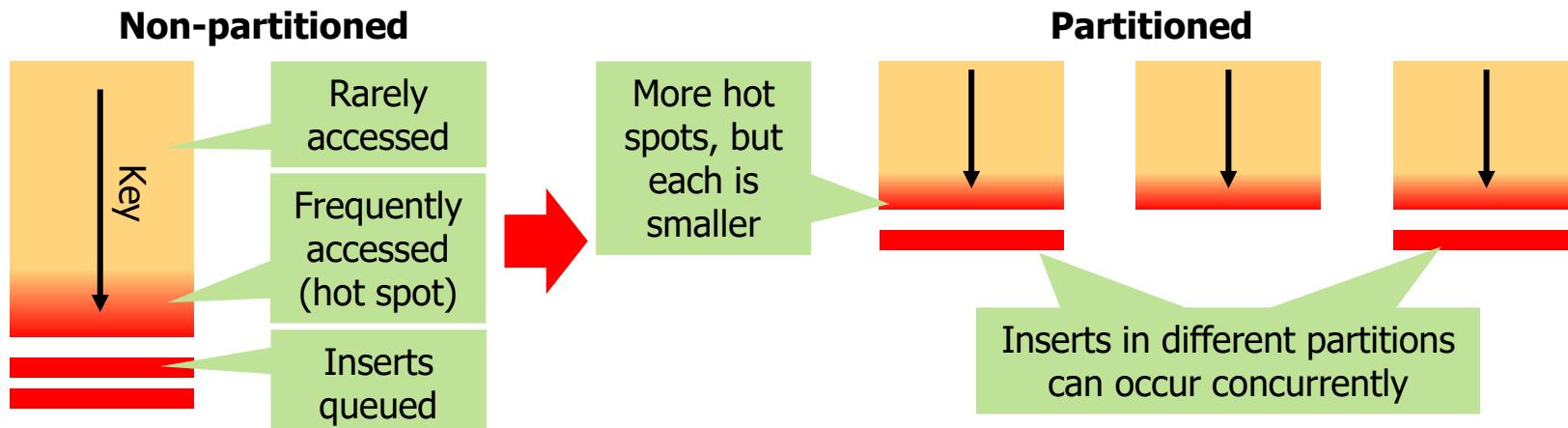
- The easiest way to scale a relational database is to add disk
- Affects throughput for data queries
 - More data to read
 - Most query optimization focuses on reducing the amount of data read
- Comes with practical considerations
 - All the data must be backed up for recovery and audit purposes
 - If the schema changes, all the data must be converted to the new schema

How Relational Databases Scale – Throughput

- Buy a bigger, faster network
 - Exotic high-speed networks connecting CPU and disk (100 Gb/s)
 - Multiple Network Interface Cards (NICs)
- Add memory to database server
 - Works for read operations in particular
 - Go to disk less often
- Design system for `INSERT` rather than `UPDATE`
 - `INSERT` is usually quicker, although it does create a “hot spot” (see next slide)

How Relational Databases Scale – Concurrency

- Buy a bigger, faster server
 - Complete each query quicker, leading to more per second
- Partition data to avoid “hot spots”
 - Common to organize tables by monotonically increasing key
 - Most operations occur on the most recent data
 - Partition data by a different factor (e.g., regional, departmental) to spread activity



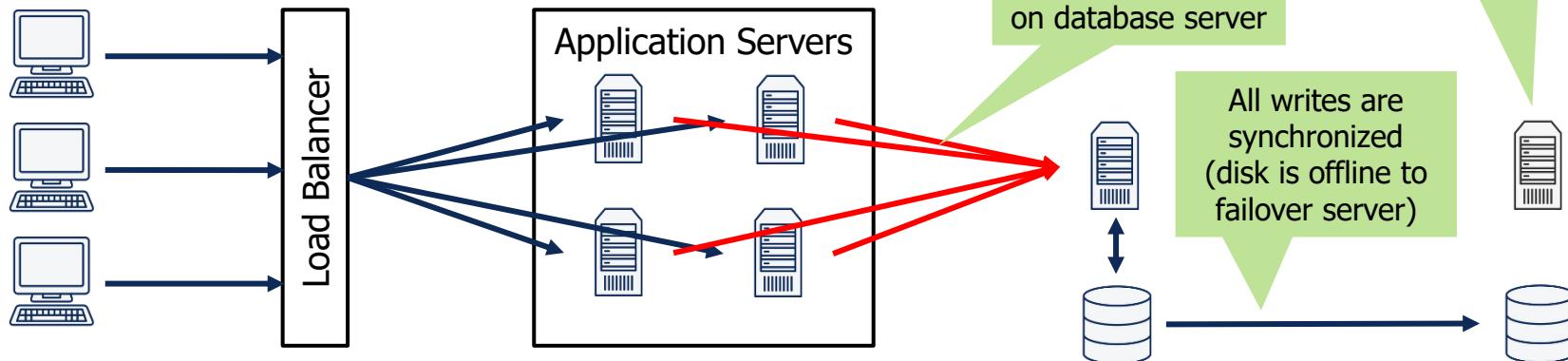
How Relational Databases Scale – Resilience

- In most systems, the database is a single point of failure
 - Mitigate by using a clustered file system with a remote replica
 - Transfer logs off-site to allow point in time recovery
- Failover times
 - Hard to achieve an active-active design, even “hot failover”
 - Usually active-passive with a manual failover

In the event of site or server failure, connect disk and switch to this server

Activity converges on database server

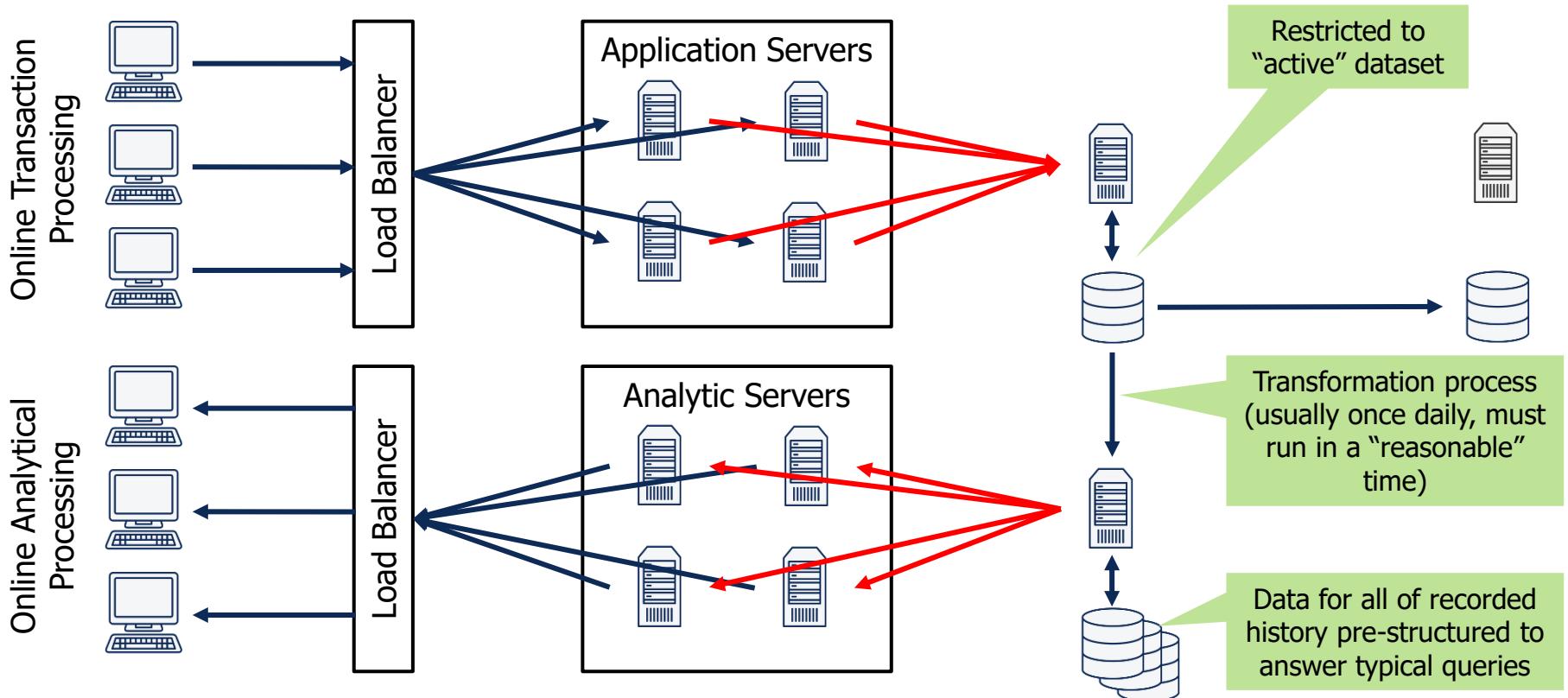
All writes are synchronized (disk is offline to failover server)



How Relational Databases Scale – Compute

- Not usually a problem for relational databases
 - The server is usually memory- and/or IO-bound
 - If it is a problem, buy a bigger box
- Bigger, better servers usually support more memory and faster networks
 - Servers are often over-specified in compute to allow them to scale other factors

Typical Database Architecture (OLTP-OLAP)



How Relational Databases Scale – Summary

- Acceptably well up to certain storage sizes
 - Terabyte level
 - A few million rows
- Beyond that, they can be made to scale with time, money, and careful design
 - Within limits
- Scaling is usually not elastic
 - To achieve elastic scaling, use cloud-based virtualization
 - Probably will not achieve the highest raw performance
- Alternatively, forget elasticity and buy an appliance
 - Highly tuned hardware and software
 - E.g., Oracle Exadata and Teradata
 - Expensive, require dedicated support

Chapter Concepts

Scaling Relational Databases

Relational Databases in the Cloud

Chapter Summary

Relational Database on IaaS

- Possible to install and run any database engine on Infrastructure as a Service
 - Amazon EC2
 - Google Compute Engine
- Advantages
 - Management is very similar to in-house
 - Full control
 - Porting most code should be straightforward
 - No effort at all unless it interacts with the environment
 - Can specify “exotic” hardware and network configurations
- Disadvantages
 - Management is very similar to in-house
 - Full responsibility
 - Not as elastic as DBaaS (next slide)

Database as a Service (DBaaS)

- Cloud operator takes care of all routine activity
 - Provisioning
 - Upgrading
 - Backup and recovery
 - Failure detection
- Advantages
 - Elastic resource allocation, pay for what you use
 - Simplified management
- Disadvantages
 - Size (AWS 16TB, GCP 10TB) and resource limits
 - Fixed choice of database engine (not a problem if yours is on the list)
 - AWS: Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle, Microsoft SQL Server
 - Google Cloud: MySQL and PostgreSQL

MySQL and PostgreSQL

- MySQL is one of the most popular cloud-based database engines
 - Owned by Oracle, but maintained as open-source
 - MariaDB is an API-compatible fork of MySQL, guaranteed to remain open-source
 - Not standards-compliant in most respects
 - However, procedural language largely follows SQL/PSM (the standard)
 - Comparison:
https://docs.oracle.com/cd/E12151_01/doc.150/e12155/oracle_mysql_compared.htm
- PostgreSQL is a popular open-source database engine
 - Maintained by a consortium
 - Emphasizes standards-compliance, but also supports extensions
 - Supports a wide range of procedural languages
 - Most widely used is PL/pgSQL, which is somewhat similar to PL/SQL (no packages)
 - Manual section on porting: <https://www.postgresql.org/docs/11/plpgsql-porting.html>
 - SQL/PSM is also supported through the optional PL/pgPSM module

Differences Between Oracle, MySQL, and PostgreSQL

■ Authentication and authorization

- MySQL supports an additional parameter so access is only allowed from defined hosts
- Oracle privileges are user- and role-based, MySQL is just user-based
- PostgreSQL also supports row-level privileges, Oracle supports this through a package

■ Schema objects

- MySQL does not have the concept of a standalone SEQUENCE, only IDENTITY columns
- Names are case sensitive in MySQL when it runs on a case-sensitive operating system
- By default, MySQL quotes identifiers with backtick (`) rather than double quotes (`")

■ Data Types

- PostgreSQL supports a wider range of data types (including IP addresses and JSON)
- Oracle DATE contains a date and a time
- Maximum size of data types differs (e.g., VARCHAR is 4,000 in Oracle, 65,532 in MySQL, 1GB in PostgreSQL)

Chapter Concepts

Scaling Relational Databases

Relational Databases in the Cloud

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The factors involved in scaling relational databases
- How relational databases can be deployed to cloud systems

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 19: Big Data and NoSQL

Chapter Overview

In this chapter, we will explore:

- The concept of Big Data
- Four leading types of NoSQL data stores
- Current developments and the future of distributed data stores

Chapter Concepts

Big Data

Introduction to NoSQL

Types of NoSQL Data Stores

NewSQL

Chapter Summary

What Is Big Data?

- Defining Big Data is not so easy—consider these definitions:

Oracle: Big Data is the derivation of value from traditional relational database-driven business decision making, augmented with new sources of unstructured data.

Microsoft: Big Data is the term increasingly used to describe the process of applying serious computing power—the latest in machine learning and artificial intelligence—to seriously massive and often highly complex sets of information.

Intel: Big Data opportunities emerge in organizations generating a median of 300 terabytes of data a week. The most common forms of data analyzed in this way are business transactions stored in relational databases, followed by documents, email, sensor data, blogs, and social media.

Technical Challenges of Big Data

■ To be categorized as Big Data, the data spans four dimensions:

- Volume
 - Ever-growing data of all types
 - Almost every activity we perform generates digital data
- Velocity
 - The rate at which data is produced
 - Growing exponentially
- Variety
 - Any type of data, structured and unstructured
 - Text, video, audio, log files, etc.
 - All data sources have potential value
- Veracity
 - Need to be able to trust the data you work with
 - Vital if we are to use it for informed business decision making

■ What challenges do these pose for traditional systems?

Where Is Big Data Relevant?

■ What are high-volume, high-velocity, high-variety situations?

- Website logs
 - Time and location of access
 - Access attempts
- Money deposits, transfers, and withdrawals
- Integrate data from different divisions and external sources

■ Typical use cases:

- Customer management
 - Gain new customers, upsell to existing customers
 - Detect fraud
- Stress tests
 - Manage risk
 - What-if analysis
- Improve workflows
- Invent new businesses

Big Data vs. Relational Database

- Big Data and SQL databases are meant for different types of problems

Criterion	Big Data	SQL Database
Scalability	To terabytes, petabytes	To gigabytes, low terabytes
Way to scale	Add more machines	Replace by more powerful machine
Type of data	Unstructured, semi-structured	Highly structured
Type of queries	Many types depending on tool sets used: we talk about them later in the course	Declarative queries: it is the job of the database to figure out the best way to compute the result of a query
Type of processing	Read mainly	Online transactions
Access path	Exploratory, unconstrained by how the data is stored	Answer pre-defined queries, constrained by the structure

Big Data and Processing

- Big Data can be categorized as:
 - Streaming
 - Data is flowing/changing
 - For example: sensor data, network data, stock market price feed, social media
 - Static
 - Long-term storage of streamed data
 - For example: web logs, emails, stock market trades
- Processing requirements are different for the above
 - Streaming requires real time processing
 - Static can be batch processed on the large data sets

Chapter Concepts

Big Data

Introduction to NoSQL

Types of NoSQL Data Stores

NewSQL

Chapter Summary

The Rise of Not only SQL (NoSQL)

- The term NoSQL is somewhat of a misnomer
 - Tends to draw attention to the query language and away from the underlying *data model*
- A *data model* organizes data elements and standardizes how the data elements relate to one another
 - See: https://en.wikipedia.org/wiki/Data_model
- Using the right data model can result in questions being answered more quickly
- A data model should be selected that most directly supports the questions being asked
 - This has to be balanced with the realities of how the data is structured when it first arrives to be processed
- Data for these stores is typically:
 - Very large data sets
 - Has no fixed schema
 - Often high-speed reads required
 - Disk- or RAM-based

BASE vs. ACID Properties

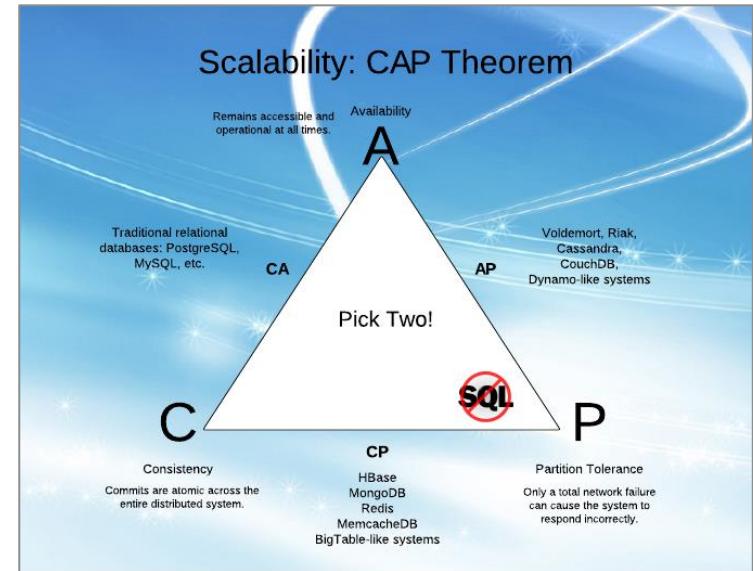
- Relational databases guarantee ACID (atomic, consistent, isolated, durable) properties
 - Since this avoids data corruption, they are seen as repositories of “truth”
 - Implementing ACID on a single-computer system is well understood
 - But NoSQL stores will *shard* data across many machines
- NoSQL stores are often said to be BASE:
 - **B**asically **A**vailable
 - **S**oft state
 - **E**ventual consistency
 - See: https://en.wikipedia.org/wiki/Eventual_consistency
- The term BASE was created in order to highlight the differences and trade-offs between two database design philosophies
 - ACID databases focus on consistency
 - BASE databases emphasize availability
 - See: <http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>

Horizontal Scalability Using Shards

- Clusters of computers are the commonly accepted way of dealing with velocity and volume
 - A divide and conquer approach
 - Break the structured data into small chunks (*database shards*)
 - Have each computer process their portion of data
 - The more computers, the smaller the shards, the faster the questions can be answered
- Where should data be stored when it is not being processed?
 - Common practice is to store the shards permanently with the computer in the cluster that is going to process this data (saves time)
 - As opposed to storing all the data in a central database
- What about algorithms and frameworks?
 - We choose those that make the most efficient use of the cluster

Enter the CAP Theorem

- In theoretical computer science, the *CAP theorem*, also known as *Brewer's theorem*, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
 - *Consistency* (C)
 - All nodes see the same data at the same time
 - *Availability* (A)
 - A guarantee that every request receives a response (success or failure)
 - *Partition tolerance* (P)
 - System continues to operate despite arbitrary message loss or failure of some part
- See: https://en.wikipedia.org/wiki/CAP_theorem
- Provides a conceptual framework for understanding differences between DBMS architectures



PACELC

- An extension to the CAP theorem
 - Described by Daniel J Abadi from Yale University in a 2010 blog post
 - Aims to unify CAP with existing knowledge around single-node RDBMSes
- The name is an acronym based on the usual formulation:
 - In the case of a Partition (P)
 - A system must choose between Availability (A) and Consistency (C)
 - Else (E) if there is no partition
 - A system must still choose between Latency (L) and Consistency (C)
- The significance lies in understanding that, even without a partition, there is still a trade-off between latency and consistency

Chapter Concepts

Big Data

Introduction to NoSQL

Types of NoSQL Data Stores

NewSQL

Chapter Summary

NoSQL Data Models

- Four different data models have distinguished themselves in the NoSQL eco-system:
 - Key-value
 - Document
 - Graph
 - Column-family
- Many data stores cover more than one storage model
- In this course, we will look quickly at the four types before covering Amazon DynamoDB in more detail in the next chapter
 - DynamoDB is a key-value and document store

Key-Value Data Stores

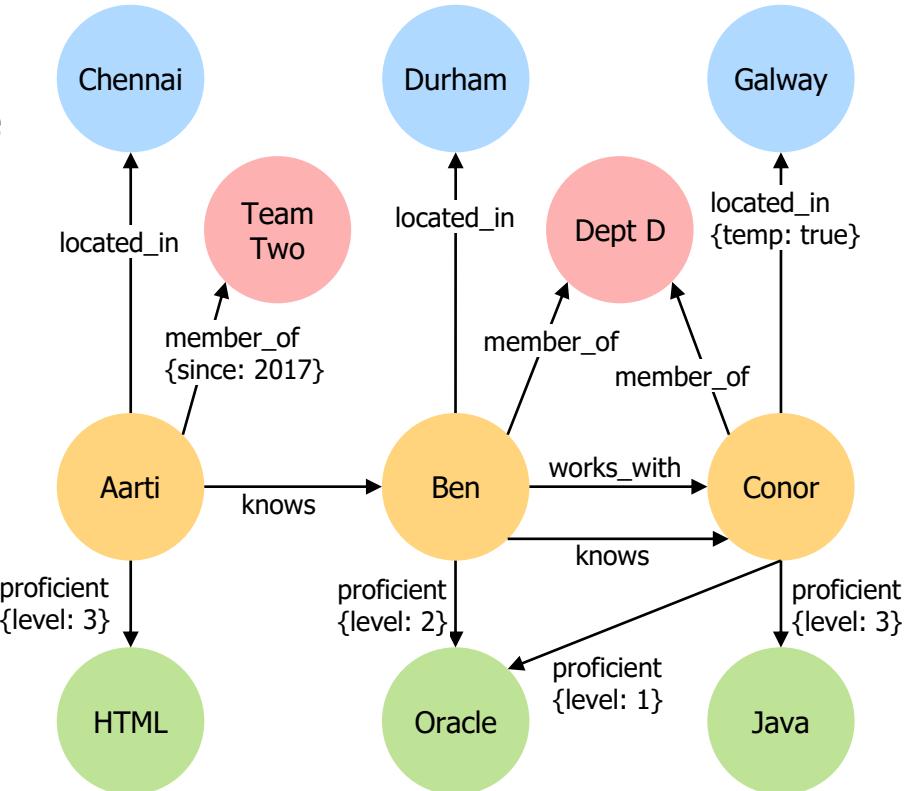
- Simplest of all the data stores
 - Key = Lookup
 - Value = Returned Information
 - May be a data structure rather than a scalar
 - Relationships cannot usually be modeled, except by convention
 - E.g., write code so that a particular field in structure is interpreted as another key
- Many products are available
 - Redis
 - Memcached
 - DynamoDB
 - Ehcache
- Typical use cases:
 - Distributed cache
 - Shared queue

Document Data Stores

- Replace concept of row with more flexible model
 - No fixed schema, every record potentially different
 - Each record is a document, usually a JSON data structure
 - Documents can be queried, e.g., by key value or contents
 - Although relationships can be modeled, this is not a strength of document stores
- Many document data stores are available
 - MongoDB
 - AWS DocumentDB (new in 2019)
 - AWS DynamoDB
 - CouchBase
- Use cases:
 - User profile management
 - Content management
 - Anywhere content is user-generated and may be of many different types

Graph Data Stores

- Allow storage of highly connected data
 - Relationships are first class members and can hold data, unlike in a relational database
 - Usually implement ACID transactions
- Entities and relationships can have types
 - Does not normally constrain their structure
 - They are usually flexible documents
- Some examples:
 - Neo4j
 - OrientDB
 - AWS Neptune
- Use cases:
 - Recommendation engine
 - Fraud detection
 - Knowledge graph



Wide-Column Data Stores

- Organize data in rows with arbitrary columns
 - Billions of rows, thousands of columns
 - Columns may be organized into families that are used together
 - Query by row key, joins are usually not supported
 - Sometimes characterized as key-value stores with wide values
 - Very high throughput
 - Distinct from column-oriented databases
 - Optimization of RDBMS, store columns together, allowing high compression
- Examples include:
 - Cassandra
 - HBase
 - Bigtable
- Use cases:
 - Real-time data capture and querying (e.g., log management, IoT)

Chapter Concepts

Big Data

Introduction to NoSQL

Types of NoSQL Data Stores

NewSQL

Chapter Summary

NewSQL

- A new generation of Relational Databases
 - Built from the ground up to support distributed ACID transactions
 - NewSQL coined to compare with NoSQL
- Typical applications are Online Transaction Processing (OLTP):
 - Short-lived transactions
 - Touch a small subset of the data
- Because of this, they rarely encounter the serious drawbacks of a distributed data store
 - Most operations take place within a single shard
- Examples:
 - Google Cloud Spanner
 - MemSQL
 - VoltDB (created by Michael Stonebraker, the originator of PostgreSQL)
 - Apache Ignite
 - CockroachDB

Google Cloud Spanner

- Just as with BigTable and Wide Column Stores, many NewSQL databases owe their design to a 2012 Google paper describing Spanner.
 - <https://ai.google/research/pubs/pub39966>
 - *“We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.”*
- Features
 - Uses the Paxos Consensus algorithm
 - Hardware-based clock synchronization (gives it the edge over competitors)
 - Each table must have a primary key (best practice anyway)
- What of CAP? It still applies
 - Spanner does pay a latency penalty for accessing shards
 - Mostly behaves like a CA system, but will sometimes sacrifice A to achieve C
 - Still achieves 5 9s availability (99.999%)

Chapter Concepts

Big Data

Introduction to NoSQL

Types of NoSQL Data Stores

NewSQL

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The concept of Big Data
- Four leading types of NoSQL data stores
- Current developments and the future of distributed data stores

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 20: Amazon DynamoDB

Chapter Overview

In this chapter, we will explore:

- Amazon DynamoDB as an example of a NoSQL store
- How to interact with DynamoDB at its native (low) level
 - At the command line
 - In Java
- How the Java Object Mapper allows a more object-oriented interaction

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

What Is DynamoDB?

- “Amazon DynamoDB is a fully-managed NoSQL database service that provides fast and predictable performance with seamless scalability.”¹
 - “Fully-managed” because operational issues such as scaling are handled automatically
 - “Fast... performance” means single-digit millisecond response
 - “Predictable performance” since it will manage performance to a defined throughput
- In addition, DynamoDB offers:
 - High availability
 - All data is automatically replicated within a region
 - ACID transactions
 - Unusual for this type of data store, not the default access mode
 - Global tables
 - By default, tables exist independently in different AWS regions
 - Encryption of data at rest
 - All stored data can be automatically encrypted

¹ <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>

Working with DynamoDB

■ AWS Console

- Management interface for all AWS Services
- Allows full access to data, as well as managing databases
- Ad hoc interactive operations, no stored programs

■ AWS CLI

- Command line interface available on Windows, Mac, and Linux
- Also allows full management and data access
- Ad hoc interactive operations or embedded in scripts

■ DynamoDB API

- Application programming interfaces available for many languages
 - Including Java and JavaScript
- Offers various options including object-based and low-level access to DynamoDB data
- Programmatic access

Accessing DynamoDB

- DynamoDB is available as a local version
 - Installed on developer workstation or accessed via Maven
 - Useful for development and testing, not suitable for production
 - No costs incurred and no network connection required
 - Requires dummy AWS credentials
 - Access using additional parameter in CLI or different connection builder in Java
 - Some behavioral differences
 - Asynchronous operations like CreateTable complete immediately
 - Reads may appear strongly consistent, in reality they are eventually consistent
 - No parallel operations
- Using Web Service version
 - Is accessed over http(s)
 - Requires real AWS credentials generated through AWS IAM console
 - Ensure code works with asynchronous operations and eventual consistency

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

Tables, Items, and Attributes

- DynamoDB stores data in **tables** similar to a relational database
 - A collection of related data
- Each table contains zero or more **items**
 - An item is similar to a row in a relational database
 - A uniquely identifiable group of attributes – identified by a pre-defined primary key
- Each item is composed of one or more **attributes**
 - An attribute is similar to a column in a relational database
 - Apart from the primary key, each item in a table can have a different set of attributes
 - Attributes may be scalar (holding a single value) or nested (containing other attributes)

Data Types and Primary Keys

- Items can have many different data types
 - Scalar: a single value; number, string, binary, boolean, and null
 - Document: structure with nested attributes; list, map
 - Set: multiple scalar values; number set, string set, binary set
 - A list may contain values of different types, a set contains items of the same type
- Every table must have a primary key defined
 - Unique value for every item in the table
 - Two forms:
 - Partition key (simple): a single attribute
 - Partition key and sort key (composite): two attributes
 - Both parts of a key may only be scalar number, string, or binary
 - Partition key, also known as hash attribute, determines how the items are stored
 - Aim to have good distribution of values and avoid hot spots
 - Sort key, aka range attribute, determines how items are organized within a partition
 - Aim to have related items stored together

Create a Table

Use the CreateTable action

- Pass parameters as shown
- TableName, KeySchema, and AttributeDefinitions are mandatory
- ProvisionedThroughput is required if using provisioned mode, more on this later

Note that only attributes in the key are defined

The action returns JSON describing the table

- Can also use DescribeTable to get this

Other table actions include:

- DeleteTable
- ListTables

String (S), Number (N),
or Binary (B)

```
{  
    "TableName": "Music",  
    "KeySchema": [  
        {  
            "AttributeName": "Artist",  
            "KeyType": "HASH"  
        },  
        {  
            "AttributeName": "SongTitle",  
            "KeyType": "RANGE"  
        }  
],  
    "AttributeDefinitions": [  
        {  
            "AttributeName": "Artist",  
            "AttributeType": "S"  

```

Insert Data

Use the PutItem action

- This is an “upsert” (updates if key matches, inserts if not)
- Returns old values if `ReturnValues` is set to `ALL_OLD` (`default` is `NONE`)

String Set

```
{  
    "TableName": "Music",  
    "Item": {  
        "Artist": {  
            "S": "Marisa Monte"  
        },  
        "SongTitle": {  
            "S": "Amor I Love You"  
        },  
        "AlbumTitle": {  
            "S": "Memorias, Cronicas e Declaracoes de Amor"  
        },  
        "Year": {  
            "N": "2000"  
        },  
        "LengthInSeconds": {  
            "N": "191"  
        },  
        "Genres": {  
            "SS": ["MPB", "Pop Rock"]  
        },  
        "Available": {  
            "BOOL": true  
        }  
    "ReturnValues": "ALL_OLD",  
}
```

Reading Data

Use GetItem action

- Retrieves a single item by full key
- Full key must be specified

```
{  
    "TableName": "Music",  
    "Key": {  
        "Artist": { "S": "Marisa Monte" },  
        "SongTitle": { "S": "Amor I Love You" }  
    }  
}
```

Use Query action

- Retrieves a collection of items
- Must match on partition key
- May specify a comparison for the sort key

```
{  
    "TableName": "Music",  
    "KeyConditionExpression": "Artist = :artist",  
    "ExpressionAttributeValues": {  
        ":artist": { "S": "Marisa Monte" }  
    }  
}
```

Use Scan action

- Scans whole table contents (slow)
- Allows arbitrary queries (non-key attributes)

```
{  
    "TableName": "Music",  
    "FilterExpression": "LengthInSeconds < :length",  
    "ExpressionAttributeValues": {  
        ":length": { "N": "180" }  
    }  
}
```

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

Using the AWS CLI

General format of commands is:

```
aws dynamodb <action> <options>
```

- For example:

```
aws dynamodb create-table --endpoint-url http://localhost:8000 \  
    --region eu-west-1 --cli-input-json file://CreateTable.json
```

Line continuation.
Use ^ in Command Tool,
' in PowerShell.

Action names use kebab-case rather than upper camel case

Core options

- When using local DynamoDB, the **--endpoint-url** is required
- **--region** is always required unless set with `aws configure`
 - `eu-west-1` = Ireland
 - `us-east-1` = North Virginia
 - `ap-south-1` = Mumbai

AWS CLI Options

- The CLI provides a number of ways to specify parameters
 - Using JSON (either in files or in a string)
 - Using individual command line options (with shorthand syntax or JSON)
- Full command in JSON
 - `--generate-cli-skeleton input` prints a JSON structure for the command
 - `--cli-input-json file://CreateTable.json`
 - Use JSON in either an input file or command line string
- Individual command line options with shorthand syntax:

```
aws dynamodb create-table --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1
```

CLI Useful Documents

■ API Reference

- <https://docs.aws.amazon.com/amazondynamodb/latest/APIReference>Welcome.html>

■ DynamoDB Developer Guide

- <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>
- <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithDynamo.html>

■ AWS CLI Command Reference

- <https://docs.aws.amazon.com/cli/latest/reference/dynamodb/index.html>



Exercise 20.1: Access DynamoDB Using AWS CLI

20 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

DynamoDB Java API

- In the AWS SDK for Java v1, there are effectively three ways to interact with DynamoDB
 - Low-level API that treats everything as a Map (`AmazonDynamoDB`)
 - Higher-level API using an `Item` class (`DynamoDB`), “the document API”
 - Object API using annotated domain classes (`DynamoDBMapper`)
- Each of these has individual benefits and drawbacks
 - Low-level API requires extensive mapping code to Java objects
 - Low-level API requires management of DynamoDB operations
 - The document API simplifies management of DynamoDB operations
 - Object API has limited support for standard Java classes and complex mapping options
 - Crucially, it is hard to convert between the different representations

Low-Level Java API

- Entry point is AmazonDynamoDB:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(
        "http://localhost:8000", "eu-west-1"))
    .build();
```

- Most actions can be performed directly from the entry point
- CreateTable and DeleteTable are asynchronous operations
 - Necessary to wait until the table is in the right state before continuing
 - AmazonDynamoDB#waiters() provides a non-blocking mechanism for waiting
- Items are represented by a Map<String, AttributeValue>
 - Must be converted to Java objects, only the key fields are guaranteed to be present
- Operations that may return large amounts of data are “windowed”
 - Return data up to a limit, which may be set in the request
 - Subsequent calls can pass in a start point

Standard Form of Low-Level Java API

API actions are invoked in the following form:

- `[Action]Result = AmazonDynamoDB# [action] ([Action]Request)`
 - E.g., `CreateTableResult result = client.createTable(request)`
- The structure of the request and response objects matches the CLI JSON objects
- There are also shorthand versions that bypass the request object

Most API objects have the following methods:

- A no-args constructor
- A number of constructors offering a shorthand way of creating “standard” objects
- Standard getters and setters (e.g., `getTableName()`, `setTableName()`)

```
CreateTableRequest request = new CreateTableRequest();
request.setTableName(TABLE_NAME);
```

- A fluent API (e.g., `withTableName()`)

```
CreateTableRequest request = new CreateTableRequest()
    .withTableName(TABLE_NAME);
```

Low-Level API Useful Resources

- Javadoc for the AmazonDynamoDB entry point
 - <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html?com/amazonaws/services/dynamodbv2/AmazonDynamoDB.html>
- Java SDK Developer Guide
 - <https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/examples-dynamodb.html>
- Java SDK Code Examples
 - <https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java>
 - https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java/example_code/dynamodb
 - Especially aws/example/dynamodb and com/amazonaws/codesamples/lowlevel

Java Document API

- The low-level API mimics the CLI, but requires boilerplate code
 - Waiting for asynchronous operations
 - Iterating over windowed results
 - Translating between Map and Java object
- Document API provides a higher level of abstraction to reduce boilerplate in many cases
- The entry point is DynamoDB:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
    .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(  
        "http://localhost:8000", "eu-west-1"))  
    .build();  
DynamoDB dynamoDb = new DynamoDB(client);
```

Working with Tables in the Document API

- The Document API provides the Table class as an entry point for table operations

Table object returned from the create operation

Returns an object to work with a table, but does no network operation, so it does not check that the table exists

```
Table table = dynamoDb.createTable(TABLE_NAME,  
    Arrays.asList(new KeySchemaElement("artist", KeyType.HASH), // Partition key  
                 new KeySchemaElement("songTitle", KeyType.RANGE)), // Sort key  
    Arrays.asList(new AttributeDefinition("artist", ScalarAttributeType.S),  
                 new AttributeDefinition("songTitle", ScalarAttributeType.S)),  
    new ProvisionedThroughput(1L, 1L));  
try {  
    table.waitForActive();  
} catch (InterruptedException e) {  
    throw new DatabaseException(e);  
}
```

Convenience methods to wait for the asynchronous operations to complete

```
Table table = dynamoDb.getTable(TABLE_NAME);  
table.delete();  
try {  
    table.waitForDelete();  
} catch (InterruptedException e) {  
    throw new DatabaseException(e);  
}
```

Working with Items in the Document API

- Items are represented by an Item class

```
private Music createMusicFromItem(Item item) {  
    Music music = new Music();  
    music.setArtist(item.getString("artist"));  
    music.setSongTitle(item.getString("songTitle"));  
    if (item.isPresent("available") && !item.isNull("available")) {  
        music.setAvailable(item.getBoolean("available"));  
    }  
    if (item.isPresent("albumTitle") && !item.isNull("albumTitle")) {  
        music.setAlbumTitle(item.getString("albumTitle"));  
    }  
    if (item.isPresent("genres") && !item.isNull("genres")) {  
        music.setGenres(new ArrayList<String>(item.getStringSet("genres")));  
    }  
    if (item.isPresent("lengthInSeconds") &&  
        !item.isNull("lengthInSeconds")) {  
        music.setLengthInSeconds(item.getInt("lengthInSeconds"));  
    }  
    if (item.isPresent("year") && !item.isNull("year")) {  
        music.setYear(item.getInt("year"));  
    }  
    return music;  
}
```

Still have to handle optional values

String Set, returned as Set<String>

Numeric attributes are returned as numbers

Standard Form of Document API

- There are a number of different forms for the Document API

- Some methods return an appropriate accessor object
- Some accept standard request objects as with the low-level API

Table object can be used to access subsequent operations

```
Table table =  
dynamoDb.createTable(request)
```

CreateTableRequest object

- Others use a spec in place of request and/or outcome in place of response
 - Usually because the request or response structure is not appropriate for use with the Document API objects

Item representation differs from PutItemResponse

```
PutItemOutcome pio = table.putItem(spec);
```

PutItemSpec because Item representation differs from PutItemRequest

- Again, there are shorthand versions that bypass the request or spec object

- API objects are similar to the low-level API with multiple constructors, getters, setters, and a fluent API

Iterating Over Windowed Results

- Methods that might return large amounts of data return special paged collections
 - ItemCollection<T> where T is a result or outcome class that contains Item
 - Implements Iterable<Item>
 - TableCollection<T> where T is a result or outcome class that contains Table
 - Implements Iterable<Table>
- Allows iteration over the whole collection
 - When iteration reaches the end of the window, the next results are seamlessly fetched
 - Need to be aware that some next() calls will cause a network operation

```
TableCollection<ListTablesResult> list = dynamoDb.listTables();
Iterator<Table> iterator = list.iterator();
while (iterator.hasNext()) {
    Table table = iterator.next();
    // process the table
}

ItemCollection<ScanOutcome> items = table.scan(spec);
Iterator<Item> iterator = items.iterator();
List<Music> music = new ArrayList<>();
while (iterator.hasNext()) {
    music.add(createMusicFromItem(iterator.next()));
}
```

Document API Useful Resources

■ Javadoc for the DynamoDB entry point

- <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html?com/amazonaws/services/dynamodbv2/document/DynamoDB.html>

■ DynamoDB Developer Guide

- <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.html>
- There is Java Document API information scattered throughout the guide
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.html>
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html>
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html>
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html>
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AppendixSampleDataCodeJava.html>

■ Java SDK Code Examples

- <https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java>
- https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java/example_code/dynamodb
 - Especially com/amazonaws/codesamples/document



HANDS-ON
EXERCISE

10 min

Exercise 20.2: Java Document API

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

Java Object Mapper

- Maps between DynamoDB Item and Java domain classes using annotations
 - Removes the need for mapping code
- The entry point is DynamoDBMapper

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(
        "http://localhost:8000", "eu-west-1"))
    .build();
DynamoDBMapper mapper = new DynamoDBMapper(client);
```

- The mapper provides simple methods to persist and restore Java objects, e.g.:
 - mapper.load
 - mapper.save
 - mapper.query
 - mapper.scan

Domain Class Annotations

- `@DynamoDBTable`
 - Links the Java class to a table
- `@DynamoDBHashKey`
 - Sets the partition key
- `@DynamoDBRangeKey`
 - Sets the sort key
- `@DynamoDBAttribute`
 - Maps an attribute to a Java property
 - Not needed if the names are the same
- `@DynamoDBIgnore`
 - Property that should not be mapped
- Property annotations may be applied to the field declaration, getter, or setter

```
@DynamoDBTable(tableName="MusicJava")
public static class MusicItem {

    @DynamoDBHashKey(attributeName="artist")
    private String artist;

    @DynamoDBRangeKey(attributeName="songTitle")
    private String songTitle;

    @DynamoDBAttribute(attributeName="genres")
    private List<String> genreList;

    @DynamoDBIgnore
    private String dummy = "DUMMY";

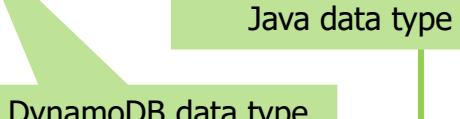
    ...
}
```

Data Type Mappings

- The mapper supports a number of core Java primitives and classes
 - String, Java primitives, and their wrappers map as expected (`char` is not mapped)
 - Date and Calendar map to String (S) in ISO8601 format
 - Set<T> maps to the appropriate Set type depending on T
- Other classes can be converted using the `@DynamoDBTypeConverted` annotation

```
@DynamoDBTypeConverted(converter=GenreConverter.class)
@dynamodbattribute(attributeName="genres")
private List<String> genreList;
```

```
public static class GenreConverter implements DynamoDBTypeConverter<Set<String>, List<String>> {
    @Override
    public Set<String> convert(List<String> object) {
        return new HashSet<String>(object);
    }
    @Override
    public List<String> unconvert(Set<String> object) {
        return new ArrayList<String>(object);
    }
}
```



Basic Mapper Operations

Load by full key

```
Music item = mapper.load(Music.class, artist, songTitle);
```

Save (insert or update)

```
mapper.save(item);
```

Query by partial key

```
Map<String, AttributeValue> values = new HashMap<>();
values.put(":artist", new AttributeValue(artist));

DynamoDBQueryExpression<Music> queryExpression =
    new DynamoDBQueryExpression<Music>()
        .withKeyConditionExpression("artist = :artist")
        .withExpressionAttributeValues(values);

List<Music> items = mapper.query(Music.class, queryExpression);
```

Scan by expression

```
Map<String, AttributeValue> values = new HashMap<>();
values.put(":length", new AttributeValue().withN("180"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("lengthInSeconds < :length")
    .withExpressionAttributeValues(values);

List<Music> items = mapper.scan(Music.class, scanExpression);
```

Object Mapper Useful Resources

- Javadoc for the DynamoDBMapper entry point
 - <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html?com/amazonaws/services/dynamodbv2/datamodeling/DynamoDBMapper.html>
- DynamoDB Developer Guide
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.html>
 - <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.html>
- Java SDK Code Examples
 - <https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java>
 - https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/java/example_code/dynamodb
 - Especially com/amazonaws/codesamples/datamodeling



HANDS-ON
EXERCISE

20 min

Exercise 20.3: Java Object Mapper

- Please complete this exercise in your Exercise Manual

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

Provisioned vs. On-Demand

- We introduced DynamoDB by talking about predictable performance; we said:
 - “Predictable performance” since it will manage performance to a defined throughput’
- There are two ways to manage the cost-performance trade-off in DynamoDB
 - On-Demand Mode
 - Pay-per-request and instantly scales to double the peak traffic of the last 30 minutes
 - A good choice if you cannot predict application traffic levels
 - Provisioned Mode (default)
 - This represents a guaranteed throughput level
 - Can be auto-scaled between a minimum and maximum value
- Requests over the limit are throttled
- Capacity is defined in terms of Read Capacity Units and Write Capacity Units
 - In general, units are linked to the number of requests per second
 - They are also related to the size of an individual item read or written

Secondary Indexes

- We have seen three actions to access data
 - GetItem by full key
 - Query by partition key
 - Scan for arbitrary access
 - Scan is much slower than the other two and should be avoided when possible
- To improve data access, we can create secondary indexes
 - Global secondary indexes have different partition key and sort key from the table
 - Created using the UpdateTable action
 - Have their own capacity limits, separate from the underlying table
 - Local secondary indexes have the same partition key and a different sort key
 - Created when the table is created
- Secondary index data
 - Indexes always contain the keys, but may also contain other attributes ("projections")
 - Indexes are sparse: only contain data where the attributes exist
- Use secondary indexes in Query or Scan action by passing in IndexName

Strongly Consistent Reads

- To achieve the defined throughput and high availability, DynamoDB tables are partitioned across availability zones
 - When values are updated, they are eventually consistent across all zones (usually < 1s)
- When values are read, they are read from one of the zones
 - Not necessarily the one that was most recently updated
 - Reads may not reflect a recent update operation
- It is possible to set a parameter on read operations to ensure strong consistency
 - "ConsistentRead": true
 - Not available on global secondary indexes
 - 1 Read Capacity Unit is 1 strongly consistent read, or 2 eventually consistent reads
- When working with a local instance of DynamoDB, all reads appear strongly consistent

Batch Operations

- Each DynamoDB action causes a network operation, so using many actions can be slow
- Batch operations group similar actions in a single network operation
 - `BatchGetItem` can contain up to 100 individual `GetItem` requests
 - `BatchWriteItem` can contain up to 25 individual `PutItem` or `DeleteItem` requests
 - Batch operations are implemented at the client level in the Java SDK (`AmazonDynamoDB`, `DynamoDB`, or `DynamoDBMapper`)
- Individual actions are invoked in parallel
- Not transactional
 - Only fail if all individual requests fail
 - Returns individual failures in `UnprocessedKeys` (get) or `UnprocessedItems` (write)
 - The most likely reason for failure is throttling, so if retrying, use an exponential back-off

ACID Transactions

- Unusually for a NoSQL data store, DynamoDB provides support for ACID transactions
 - As with a Relational Database, can set the isolation level for transactions
- TransactGetItems
 - Up to 10 Get requests
- TransactWriteItems
 - Up to 10 write operations, that may be Put, Update, Delete, or ConditionCheck
 - ConditionCheck checks that an item exists, or has specific attribute values
 - Supports idempotency through client tokens
 - If a new request with the same token is seen within a 10 min period, it is considered the same request being re-submitted and will not be applied twice
- Transactional operations consume twice as many units as non-transactional operations
 - Will always fail if the table does not have sufficient capacity
- Implemented at the client level in the Java SDK (AmazonDynamoDB, DynamoDBMapper)
 - Not implemented for the Document API (DynamoDB)

Exception Handling

- Most DynamoDB methods throw unchecked exceptions with one of two super-classes
 - `AmazonServiceException` indicates that the request was received by DynamoDB, but could not be processed
 - `AmazonClientException` indicates that the client could not get a response, or could not interpret the response
- In our code samples, we have allowed unchecked exceptions to propagate and cause the whole application to fail
 - In most cases, you will want to do something different
- The AWS SDK implements an automatic exponential back-off
 - Where it would be beneficial (e.g., throttling exceptions)
 - By the time client code receives an exception there is often no point in retrying again
 - Can configure this at the client level (`AmazonDynamoDB`)
 - Use a `ClientConfiguration` with a `RetryPolicy`
 - E.g., to switch to a “fail-fast” strategy

Conditional Writes

- PutItem, UpdateItem, and DeleteItem support an optional ConditionExpression
 - Operation will only complete if the condition is true
- Three key uses
 - Prevent overwrite
 - PutItem is an upsert (it will update an item if it already exists, insert otherwise)
 - We can check whether an item with the same key already exists
 - Make updates idempotent
 - Check that the attribute to be updated has the before value that we expect
 - Implement optimistic locking
 - Check that all attribute values are the same as when we first retrieved the Item
- In addition to standard comparison operators, supports a small set of functions
 - attribute_exists, attribute_not_exists
 - begins_with
 - contains

Connecting to a Remote Instance

- Unfortunately, connecting to a local instance and a remote instance are different
 - When connecting to a local instance, must specify the full endpoint:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
    .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(  
        "http://localhost:8000", "eu-west-1"))  
    .build();
```

- When connecting to a remote instance, the endpoint is automatically determined from the region:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
    .withRegion("eu-west-1")  
    .build();
```

- And the region can even be automatically determined from the profile:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();
```

- The embedded version is even more different:

```
AmazonDynamoDB client = DynamoDBEmbedded.create().amazonDynamoDB();
```

Streams

- DynamoDB Streams represent Item changes as a series of stream records
 - A stream is associated with a particular table
 - Records are automatically removed after 24 hours
- A stream record is written whenever one of the following events occurs:
 - A new item is added to the table, the record contains the new item
 - An item is updated, the record contains the before and after images of the item
 - An item is deleted, the record contains the item before it was deleted
- Create a stream using `CreateTable` or `UpdateTable`

KEYS_ONLY, NEW_IMAGE,
OLD_IMAGE, NEW_AND_OLD_IMAGES

```
"StreamSpecification": {  
    "StreamEnabled": true,  
    "StreamViewType": "KEYS_ONLY"  
}
```

- The Java client is different

```
AmazonDynamoDBStreams client = AmazonDynamoDBStreamsClientBuilder.standard()  
    .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(  
        "http://localhost:8000", "eu-west-1"))  
    .build();
```

Chapter Concepts

What Is DynamoDB?

Core Concepts

AWS CLI

Java API

Java Object Mapper

Advanced Topics

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Amazon DynamoDB as an example of a NoSQL store
- How to interact with DynamoDB at its native (low) level
 - At the command line
 - In Java
- How the Java Object Mapper allows a more object-oriented interaction

Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Course Summary

Course Summary

In this course, we have:

- Examined the role of SQL and the basic toolset
- Selected, filtered, and sorted data from the database
- Manipulated the data with Oracle functions
- Extracted data from multiple tables
- Aggregated data with the group functions
- Created and managed tables, views, and indexes
- Programmed with PL/SQL
- Created stored procedures, functions, and packages
- Created and worked with triggers
- Explored data quality and data movement